Fast Fourier Transforms and the Fujitsu VPP300

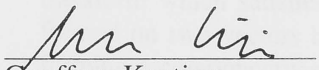
Geoffrey Keating

February 2000

A thesis submitted for the degree of Doctor of Philosophy of The Australian National University.

Declaration

This thesis is all my own original work.



Geoffrey Keating

Abstract

A one-dimensional Discrete Fourier Transform (DFT) is defined to be a linear transform which satisfies the *convolution property*: a convolution can be performed on two vectors by performing a DFT on each of the inputs, then performing a componentwise multiplication, then an inverse DFT. The importance of the DFT is that it can be performed in time $O(n \log n)$, where n is the length of the vectors, for any n. Four algorithms are presented which combine to allow this: the prime factor algorithm can be used when n can be factored into two relatively prime factors; the Cooley-Tukey algorithm can be used when n can be factored, even if the factors are not relatively prime; Rader's algorithm can be used when n is prime to reduce the problem to that of computing a longer DFT but one whose length is more convenient; and Bluestein's algorithm can be used as for Rader's algorithm but need not have n prime. In general, using the Cooley-Tukey algorithm is both faster and more accurate than simply computing the DFT as a matrix-vector product.

These algorithms are combined into a *framework* to allow the computation of many sizes of DFT. When choosing algorithms in a framework, one of the important criteria is efficiency, and one important factor determining efficiency is the pattern of data movement. In particular, there are many variations of the Cooley-Tukey algorithm which differ primarily in the pattern of data movement employed. For vector supercomputers, it is desirable to arrange the data movement in such a way as to ensure that the data movement is performed by *square transpositions*, that is transpositions in which the dimensions being transposed are of equal size, and an algorithm is presented which achieves this.

One significant application of the DFT is in the multiplication of integers. An extension of this application, by the application of group theory, allows the generalisation of convolutions and of the DFT to vectors indexed by an arbitrary group. In particular, when the group is abelian, it allows the derivation of multidimensional convolutions and the multidimensional DFT. Algorithms are presented that

allow computation of the multidimensional DFT in time $O(n \log n)$, including equivalents of the four one-dimensional algorithms listed. New algorithms are presented which can perform multidimensional DFTs using only square transpositions.

The 'square transpose' algorithms were implemented on the Fujitsu VPP300 vector supercomputer. Important details of the implementation are described, and the performance of the resulting implementation is analyzed. The new multidimensional algorithm gives significantly better performance than previous techniques. The implementation of a parallel DFT algorithm is described and its performance is analyzed; near-linear speedup is obtained in the case examined.

In many applications, it is known that the input of the DFT is real, the result is real, or both. Two different techniques are presented to take advantage of the reduced input domain to achieve substantial computational savings. The first operates by reducing two or more symmetric DFTs to a single DFT with arbitrary input; it can also be applied to reduce a single symmetric DFT to a shorter DFT with arbitrary input. The second technique examines the symmetries arising from the application of the Cooley-Tukey algorithm to a symmetric DFT, and allows the reduction of computation internally in the algorithm. A hybrid of these techniques was implemented on the VPP300 for the computation of real transforms. Sine and cosine transforms were also implemented. The implementations' performance were analyzed and compared against the complex DFT algorithms.

Preface

This thesis describes a little under three years' research and development of the Fast Fourier Transform, as part of a doctoral program at the Australian National University. Some of the work presented here has been published previously [54, 55, 56].

Some of the most significant original contributions presented in this work are:

- a unified treatment of the Prime Factor Algorithm and the Cooley-Tukey FFT Algorithm (sections 1.3, 1.4);
- a classification of the various one-dimensional FFT algorithms based on the Cooley-Tukey algorithm, including a graphical notation which describes the algorithms and allows this classification at a glance (section 1.10);
- a description of the various multidimensional FFT algorithms based on module theory, including explicit formulations of the multidimensional versions of the one-dimensional FFT algorithms (chapter 2);
- a new multidimensional FFT algorithm which uses only square transpositions (section 2.12), its implementation and a performance analysis of the implementation (chapter 3); and
- a new real FFT algorithm, a hybrid of two varieties of real FFT algorithms, with some advantages (and some disadvantages) of both, its implementation and a performance analysis (chapter 4).

I would like to thank my supervisors Markus Hegland and Lotzy Kovacs; the other members of my supervisory panel Richard Brent and Michael Osborne; Fujitsu Corporation and in particular Mr. Nakanishi; the examiners; and my parents Con and Shirley Keating. All have been essential in the production of this thesis.

Contents

List of Figures

Chapter 1

The Discrete Fourier Transform

There are a wide variety of ways to define the discrete Fourier transform. Historically, a Fourier series—a continuous relation of the DFT—was used in a paper [59] by Lagrange in 1759; in 1807, Fourier presented a paper claiming that an arbitrary function could be represented as a Fourier series, which would validate the continuous Fourier transform. The discrete Fourier transform also occurred in the late eighteenth century; a paper by Gauss [34] used it to find a trigonometric interpolation for astronomical observations [17].

This gives us two ways to approach the DFT: as a discretisation of the continuous Fourier transform, or as a particular linear transformation. The approach we will prefer will be neither of these; instead, we will focus on the most important property of the DFT, the *convolution property*.

1.1 Convolutions

A *convolution* is a function of two vectors of length n, say x and y, which we will write as $x \star y$. We define $x \star y$ by:

$$(x \star y)_j := \sum_{k=0}^{n-1} (x)_k (y)_{j-k} \tag{1.1}$$

Here, $(x)_k$ means the $k+1$th element of x. Vector indexes will be considered to 'wrap around'; that is, $(x)_{-1}$ is the nth element of x, and $(x)_{n+1}$ is the second element of x, $(x)_1$. This and other notation used is summarised in appendix A.

Convolutions are used in a wide variety of fields including signal processing and statistics. They are even used in the multiplication of natural numbers (see

section 1.13 below).

In matrix notation, the above expression is equivalent to Yx, where Y is a *circulant* matrix:

$$x \star y := \begin{pmatrix} (y)_0 & (y)_{n-1} & \cdots & (y)_1 \\ (y)_1 & (y)_0 & \cdots & (y)_2 \\ \vdots & \vdots & \ddots & \vdots \\ (y)_{n-1} & (y)_{n-2} & \cdots & (y)_0 \end{pmatrix} x \tag{1.2}$$

1.2 The Discrete Fourier Transform

We can then define a DFT as an invertible linear function F_n on vectors of length n which has the *convolution property*:

Define $x \odot y$ to be the componentwise product of x and y; then we define a DFT F to be such that for any x and y:

$$x \star y = F_n^{-1}((F_n x) \odot (F_n y)) \tag{1.3}$$

That is, to calculate a convolution, we can perform a Fourier transform of its operands, multiply them componentwise, and perform an inverse Fourier transform.

1.2.1 Roots Of Unity

To actually write DFTs explicitly, we will make the following definition: Let

$$\omega_n = \exp\left(\frac{2\pi\sqrt{-1}}{n}\right); \tag{1.4}$$

that is, ω_n is a nth root of unity, so that $\omega_n^n = 1$.

Following the usual rules for complex exponentials, we can calculate powers of ω_n as:

$$\omega_n^\alpha = \cos\frac{2\pi\alpha}{n} + \sqrt{-1}\sin\frac{2\pi\alpha}{n} \tag{1.5}$$

The complex conjugate of ω_n, $\overline{\omega_n}$, is then:

$$\overline{\omega_n^\alpha} = \cos\frac{2\pi\alpha}{n} - \sqrt{-1}\sin\frac{2\pi\alpha}{n} \tag{1.6}$$

$$= \cos -\frac{2\pi\alpha}{n} + \sqrt{-1}\sin -\frac{2\pi\alpha}{n} \tag{1.7}$$

$$= \omega_n^{-\alpha} \tag{1.8}$$

1.2.2 Some DFTs

Given the formulation in equation 1.3, any invertible linear transformation F_n (written multiplicatively) of the following form is a DFT of size n:

$$(F_n x)_j = \sum_{k=0}^{n-1} \omega^{qjk}(x)_k \tag{1.9}$$

Here, q may be any integer which is relatively prime to n; usually 1 or -1 is used. Usually, F_n will be abbreviated as F where its size is clear from the context.

To see that this transformation satisfies equation 1.3, notice that

$$\begin{aligned}
((Fx) \odot (Fy))_j &= \left(\sum_{k=0}^{n-1} \omega^{qjk}(x)_k\right)\left(\sum_{i=0}^{n-1} \omega^{qji}(y)_i\right) && (1.10)\\
&= \sum_{k=0}^{n-1}\sum_{i=0}^{n-1} \omega^{qj(k+i)}(x)_k(y)_i && (1.11)\\
&= \sum_{k=0}^{n-1}\sum_{l=k}^{k+n-1} \omega^{qjl}(x)_k(y)_{l-k} && (1.12)\\
&= \sum_{k=0}^{n-1}\sum_{l=0}^{n-1} \omega^{qjl}(x)_k(y)_{l-k} && (1.13)\\
&= \sum_{l=0}^{n-1} \omega^{qjl}\sum_{k=0}^{n-1}(x)_k(y)_{l-k} && (1.14)\\
&= F(x \star y) && (1.15)
\end{aligned}$$

The step from equation 1.12 to 1.13 uses the equalities $\omega^{qjn} = 1$ and $(y)_{l-k} = (y)_{l-k-n}$.

It is then only necessary to show that F_n is an invertible transformation; this will be done below.

In fact, equation 1.9 describes all the linear transformations that satisfy the convolution property; see section 2.4. It is important to note that multiples of F—in particular, $\frac{1}{\sqrt{n}}F$, which is commonly used in the literature—do not satisfy the convolution property as we have defined it.

1.2.3 The DFT Matrix and The Inverse Matrix

As stated above, the DFT is a linear transformation. It has an associated matrix, which we will also call F:

$$F = \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^q & \omega^{2q} & \cdots & \omega^{(n-1)q} \\ \omega^0 & \omega^{2q} & \omega^{4q} & \cdots & \omega^{(n-2)q} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{(n-1)q} & \omega^{(n-2)q} & \cdots & \omega^q \end{pmatrix} \tag{1.16}$$

where $(F)_{i,j} = \omega^{ijq}$. Matrices, like vectors, are indexed starting from zero. As can be seen, the matrix is symmetric.

Theorem 1 $F^{-1} = \frac{1}{n}\overline{F}$.

The j row of F has the form

$$(F)_j = (\omega^0, \omega^{qj}, \omega^{2qj}, \cdots, \omega^{(n-1)qj})$$

The componentwise product of the j and k rows of F is then

$$(\omega^0, \omega^{q(j+k)}, \omega^{2q(j+k)}, \cdots, \omega^{(n-1)q(j+k)})$$

that is, it is the $j+k$ row of F. The complex conjugate of the j row of F, $\overline{(F)_j}$, will be

$$(\omega^0, \omega^{-qj}, \omega^{-2qj}, \cdots, \omega^{-(n-1)qj})$$

so the complex conjugate of the j row is the $n-j$ row.

So suppose $(F)_i$ and $(F)_j$ are rows of F. Then the inner product of $(F)_i$ and $(F)_j$,

$$\langle (F)_i, (F)_j \rangle = \sum_{k=0}^{n-1} ((F)_i \odot \overline{(F)_j})_k \tag{1.17}$$

is just the sum of all the elements of $(F)_{i-j}$.

What is this sum? The following formula applies to calculating sums of powers: If $a \neq 1$,

$$\sum_{k=1}^{n} a^k = \frac{a^n - 1}{a - 1} \tag{1.18}$$

In our case, if we set $a = \omega^{q(i-j)}$, we obtain

$$\sum_{k=0}^{n-1} \omega^{q(i-j)k} = \omega^{-q(i-j)} \sum_{k=1}^{n} \omega^{q(i-j)k} \tag{1.19}$$

$$= \omega^{-q(i-j)} \frac{\omega^{q(i-j)n} - 1}{\omega^{q(i-j)} - 1} \tag{1.20}$$

$$= 0 \tag{1.21}$$

whenever $\omega^{q(i-j)} \neq 1$, that is when $q(i-j)$ is not a multiple of n. Since q is relatively prime to n, then if $q(i-j)$ is a multiple of n then $i-j$ is a multiple of n; but since $-n < i-j < n$, $i-j$ is only a multiple of n when $i = j$.

Therefore, $\langle (F)_i, (F)_j \rangle = 0$ for all i, j so that $i \neq j$, and the rows of F are orthogonal.

For each row $(F)_i$, the norm of the row is

$$||(F)_i|| = \sqrt{\sum_{j=0}^{n-1} \omega^{ij} \overline{\omega^{ij}}} \tag{1.22}$$

$$= \sqrt{n} \tag{1.23}$$

so if we divide F by $\sqrt{n}$, the rows of F will be orthonormal. Sometimes, the DFT is defined as $\frac{1}{\sqrt{n}} F$, for this reason; we have instead preferred a DFT that has the convolution property.

Since the rows of $\frac{1}{\sqrt{n}} F$ are orthonormal, $\frac{1}{\sqrt{n}} F$ must be unitary. From standard linear algebra it follows that F^{-1} exists, and

$$F^{-1} = \frac{1}{n} \overline{F}^t = \frac{1}{n} \overline{F}. \tag{1.24}$$

This is what we wanted to prove. □

Notice that from equation 1.8, $\overline{F}$ is just F with q replaced by $-q$. So nF^{-1} is also a DFT. Also notice that with our definition, F^{-1} is *not* a DFT.

1.3 The Fast (Discrete) Fourier Transform

The convolution property itself does not lead to any particular saving in computational effort; in fact, it might seem to be faster to perform a matrix-vector product for the convolution than to use the convolution property and perform three DFTs as matrix-vector products, and a componentwise multiplication. However, this

assumes that a matrix-vector product is the most efficient way to perform a DFT, which we will show is not the case.

Let r, s, t, u be integers so that $\gcd(r, s) = 1$ and $\gcd(t, u) = 1$. Then for any $0 \leq j < n$, we can write $j = ar + bs$ and $k = ct + du$ for suitable[1] ranges of a, b, c, d. We can then rewrite

$$(Fx)_j = \sum_{k=0}^{n-1} \omega^{jk}(x)_k \tag{1.25}$$

as

$$(Fx)_{ar+bs} = \sum_{ct+du=0}^{n-1} \omega^{(ar+bs)(ct+du)}(x)_{ct+du} \tag{1.26}$$

Suppose we chose $rt = n$. Then we can constrain $0 \leq d < t$ and $0 \leq c < r$. Using various properties of exponentials and summations:

$$(Fx)_{ar+bs} = \sum_{d=0}^{t-1} \omega^{adru}\omega^{bdsu} \sum_{c=0}^{r-1} \omega^{acrt}\omega^{bcst}(x)_{ct+du} \tag{1.27}$$

Then since $\omega_n^n = 1$, we have:

$$(Fx)_{ar+bs} = \sum_{d=0}^{t-1} \omega_t^{adu}\omega_n^{bdsu} \sum_{c=0}^{r-1} \omega_r^{bcs}(x)_{ct+du} \tag{1.28}$$

If we choose $s = u = 1$, we obtain the splitting formula that underlies the usual Cooley-Tukey FFT:

$$(Fx)_{ar+b} = \sum_{d=0}^{t-1} \omega_t^{ad}\omega_n^{bd} \sum_{c=0}^{r-1} \omega_r^{bc}(x)_{ct+d} \tag{1.29}$$

Note here that the inner summation is equivalent to t DFTs on vectors of length r (although not on sequential elements), and the outer summation is *almost* equivalent to r DFTs on vectors of length t. The 'almost' term here, ω_n^{bd}, is usually called the *twiddle factor*.

The above used $q = 1$ for simplicity, but the only property of ω that it requires is that $\omega_n^n = 1$; so it will be valid for any value of q.

[1] This is intentionally vague; there can be more than one 'suitable' range. An example of a suitable range for a and b is if a ranges from 0 to $\frac{n}{\gcd(n,r)} - 1$ and b ranges from 0 to $\gcd(n, r) - 1$; but this may not be very useful, for instance if $r = 1$.

By recursively computing the smaller DFTs of size r and t using this method (if r and/or t are not prime), we can reduce the number of arithmetic operations required to be

$$O\left(n\sum_{j=1}^{t}k_jp_j\right)$$

where $n=\prod_{j=1}^{t}p_j^{k_j}$ and the p_j are prime. If n is a power of p, so that $n=p^k$, we find that the number of arithmetic operations required for a fixed p is

$$O\left(n\log_p n\right).$$

This is of optimal order. [66] shows that the FFT requires at least

$$\frac{n}{2}\log_2 n \tag{1.30}$$

additions in a linear algorithm.

A method similar to this was used in [34], but the form given above is based on James W. Cooley and John W. Tukey's paper [24]. Many authors have presented a matrix version of this formula, for instance [70, 51, 28, 80, 86, 61].

1.4 The Prime Factor Algorithm

The above does not require any special properties of r and t, other than that their product is n. If we can factor n so that r and t are coprime (that is, if n is not a power of a prime), we can write every j as $j=ar+bt$ and k as $k=ct+dr$, where $a,d\in\{0\ldots t-1\}$ and $b,c\in\{0\ldots r-1\}$. Then, following on from equation 1.26 above:

$$(Fx)_{ar+bt} = \sum_{d=0}^{t-1}\sum_{c=0}^{r-1}\omega^{(ar+bt)(ct+dr)}x_{ct+dr} \tag{1.31}$$

$$(Fx)_{ar+bt} = \sum_{d=0}^{t-1}\omega^{adrr}\sum_{c=0}^{r-1}\omega^{(ac+bd)rt}\omega^{bctt}x_{ct+dr} \tag{1.32}$$

$$(Fx)_{ar+bt} = \sum_{d=0}^{t-1}\omega_t^{adr}\sum_{c=0}^{r-1}\omega_r^{bct}x_{ct+dr} \tag{1.33}$$

Here there is no twiddle factor. Also, note that the actual values of ω_t^{ads} are those of ω_t^{ad}, but in a permuted order; the above can be rewritten as

$$(F(x))_{ar+bt}=\sum_{d=0}^{t-1}\omega_t^{ad}\sum_{c=0}^{r-1}\omega_s^{bc}x_{cq+dr} \tag{1.34}$$

for some q and r. The practical computation of the indices can be performed by Euclid's algorithm.

The prime factor algorithm seems to have first appeared in [37].

A complete analysis of the possible choices of j and k in equation 1.26 is given in [41, 18]; essentially, the only possibilities are combinations of the above two algorithms. Some of the combinations use fewer powers of ω than the above algorithms, which is of benefit if they must be computed during the FFT rather than being provided as constants.

1.5 Rader's Algorithm

The previous two sections require factoring n, and then an FFT can be computed by computing FFTs of the size of the factors together with some additional computation. Eventually, however, n will be reduced to a prime number and these techniques no longer apply. In this section, we present an algorithm due to Rader [73], which deals with this specific case.

Suppose n is a prime. Then there exists at least one $1 < g \leq n-1$ so that for all $1 \leq i \leq n-1$, there exists exactly one $1 \leq j \leq n-1$ so that

$$j \equiv g^i \pmod{n}. \tag{1.35}$$

Such a g is called a *primitive root modulo* n. To find such a primitive root, it is sufficient to guess g and check that it is a primitive root; the probability of making a wrong guess is

$$\frac{1}{O(\log \log n)} \tag{1.36}$$

so it is rare that more than one or two guesses are required [57, p. 391]. Note that equation 1.35 is equivalent to saying that

$$\omega_n^j = \omega_n^{g^i}. \tag{1.37}$$

Fermat's little theorem says that for any integer $g < n$,

$$g^{n-1} \equiv 1 \pmod{n} \tag{1.38}$$

so g^{-k} can be defined to be $g^{-k} \equiv g^{n-1-k} \pmod{n}$.

Now, let us apply this to equation 1.9. The first element of the vectors must be treated specially:

$$(F_n x)_0 = \sum_{k=0}^{n-1} (x)_k \tag{1.39}$$

then for $1 \leq j \leq n-1$, we can substitute g^j for j and g^{-k} for k:

$$(F_n x)_{g^j} - (x)_0 = \sum_{k=1}^{n-1} \omega^{qg^j g^{-k}} (x)_{g^{-k}} \tag{1.40}$$

$$= \sum_{k=1}^{n-1} \omega^{qg^{j-k}} (x)_{g^{-k}} \tag{1.41}$$

$$\tag{1.42}$$

Now, compare equation 1.41 and the convolution equation 1.1. 1.41 is a convolution of the last $n-1$ elements of x permuted by g^{-1}, and the vector w of length $n-1$ defined by

$$(w)_i = \omega_n^{qg^{i+1}} \tag{1.43}$$

Then the convolution property applies and so an FFT of prime order n (that is, on vectors of length n) can be computed in the time required to compute two FFTs of order $n-1$, a summation of n elements, and a componentwise multiplication of size $n-1$, assuming that $F_{n-1}w$ is already computed.

Sometimes computing $F_{n-1}x$ can be inconvenient; for instance, computing an FFT of size 283 using Rader's method directly requires applying it to 283, 47, 23, 11, 5, and 3, assuming FFTs of size 2 are computed directly. To avoid this, the following observation can be used: A convolution $x \star y$ of length n can easily be extended to a longer convolution $x' \star y'$ where x' and y' are of length $m \geq 2n-1$, so that $(x \star y)_i = (x' \star y')_i$ for $0 \leq i < n$, by choosing

$$x'_i = \begin{cases} x_i, & \text{when } i < n; \\ 0, & \text{otherwise.} \end{cases} \tag{1.44}$$

$$y'_i = \begin{cases} y_i, & \text{when } i < n; \\ y_{i-m}, & \text{otherwise.} \end{cases} \tag{1.45}$$

Then a convolution, such as equation 1.41, whose length is a product of inconveniently large primes can be computed as a larger convolution, but one whose length is a product of smaller primes.

1.6 Bluestein's Algorithm

Bluestein's algorithm relies on the following equality:

$$jk = \frac{j^2 + k^2 - (j-k)^2}{2} \tag{1.46}$$

Substituting this into equation 1.9, we obtain

$$(F_n x)_j = \sum_{k=0}^{n-1} \omega_n^{q\frac{j^2+k^2}{2}} \omega_n^{-q\frac{(j-k)^2}{2}} (x)_k \tag{1.47}$$

$$= \omega_n^{q\frac{j^2}{2}} \sum_{k=0}^{n-1} \omega_n^{-q\frac{(j-k)^2}{2}} \omega_n^{q\frac{k^2}{2}} (x)_k \tag{1.48}$$

$$= \omega_{2n}^{qj^2} \sum_{k=0}^{n-1} \omega_{2n}^{-q(j-k)^2} \omega_{2n}^{qk^2} (x)_k \tag{1.49}$$

The last equation can be computed using a convolution $x' \star w$ of length $2n$, where

$$(x')_k = \begin{cases} \omega_{2n}^{qk^2}(x)_k, & \text{when } k < n; \\ 0, & \text{otherwise.} \end{cases} \tag{1.50}$$

$$(w)_k = \omega_{2n}^{-qk^2} \tag{1.51}$$

followed by a componentwise multiplication with $\omega_{2n}^{qj^2}$. In fact, the convolution can be of any size at least $2n$.

This allows the computation of any size FFT using two FFTs of size at least $2n$, assuming the DFT of w is precomputed.

The basic Bluestein algorithm was published in [15].

The Bluestein algorithm can be extended to efficiently compute expressions of the form

$$(y)_j = \sum_{k=0}^{n-1} \zeta^{jk} (x)_k \tag{1.52}$$

for any complex ζ, using a similar technique to the above. This technique, called the "Chirp z-transform" ("chirp" after the sound whose amplitude is $\Re(w)$) in the form described above, and various applications of it, are described in [72, 9].

1.7 FFT Frameworks

The previous sections describe the basic mathematics underlying almost all FFT implementations. To go from the mathematics to an efficient implementation, it is necessary to:

1. Choose which of the various FFT algorithms is to be used to compute each size of FFT, and the order in which they are to be applied;

2. Decide on how the data is to be represented in memory; and

3. Determine how the operations required can be efficiently implemented on a particular machine.

The way in which these choices are made for all sizes of FFT describes an FFT *framework*. For a particular size, we will call the choices a *schema*.

These choices will affect both the time and space requirements of the resulting algorithms. All of these choices are interdependent and also depend strongly on the details of the machine.

Even the question of how to manage these choices has gained some attention; [33] uses a dynamic programming algorithm, combined with runtime estimation of the machine's parameters, to determine an FFT schema, and [49, 50] proposes a design methodology and a special-purpose compiler to assist with it.

Since FFTs are so important, it is not uncommon for special-purpose hardware to be built to implement it. This topic will not be covered here.

In the following sections, we will examine more closely how these issues are addressed.

1.8 Small FFT Calculation

An FFT framework usually works by reducing a large FFT down to small DFTs, which can then be computed directly (by equation 1.9). In practise it is useful to write explicit routines for computing FFTs of small sizes, up to about 32 (depending on the machine in use), to better use the high levels of the memory hierarchy, and to allow a compiler to see more of the computation at once and so better schedule (and otherwise optimise) the operations.

These *elementary* FFT routines account for a substantial part of the CPU time involved in the computation, and various strategies have been used to optimise them.

At the lowest level, one common technique is to hand-code highly processor-specific routines in assembly language, or in high-level language statements that are intended to produce specific machine code output. This produces performance improvements by avoiding limitations in the compiler's optimiser and by encouraging the programmer to modify the algorithm to take advantage of specific architectural features [12].

At a slightly more general level, modern CPUs often have separate multiply and addition pipelines, and can perform multiplications at about the same speed

as additions; it is thus desirable to ensure that as many operations as possible are of the form

$$(y)_j = \alpha \cdot (x)_j + \beta \tag{1.53}$$

for real α, β, x and y [60, 36] or to at least try to trade off additions for multiplications where there are more additions [88]. The essential trick here for a Cooley-Tukey algorithm is to join the twiddle factor multiplication, which is a complex multiplication and thus has more multiplications than additions, with the elementary FFT computation which will have more additions than multiplications.

On older systems multiplication was sometimes much slower than addition and so it was desirable to reduce the number of multiplications required. In this case it is often convenient to rewrite a complex multiplication

$$\begin{aligned} \Re(\alpha\beta) &= (\Re(\alpha)\Re(\beta) - \Im(\alpha)\Im(\beta)) \\ \Im(\alpha\beta) &= (\Re(\alpha)\Im(\beta) + \Im(\alpha)\Re(\beta)) \end{aligned}$$

as

$$\begin{aligned} \Re(\alpha\beta) &= (\Re(\alpha) + \Im(\alpha))(\Re(\beta) + \Im(\beta)) \\ &\quad -\Im(\alpha)\Re(\beta) + \Re(\alpha)\Im(\beta) \\ \Im(\alpha\beta) &= \Re(\alpha)\Im(\beta) + \Im(\alpha)\Re(\beta) \end{aligned}$$

which requires 3 multiplications and 5 additions compared with 4 multiplications and 2 additions for the first version [61].

At the highest level, there is the question again of how to generate these elementary FFT subprograms. The traditional technique of hand-coding the subprograms is by far the most commonly used, but other techniques include generating the subprograms at compile-time through an automated process (dynamic programming [25, 26] seems to be useful here) [48, 77, 33] or even at run-time.

Of course, the elementary FFT subprograms will also encounter some of the questions discussed below for the general case, in particular the choice of which FFT algorithm to use.

It is also possible to have an FFT framework in which the FFT is computed by conversion to some other operation, for instance by converting the FFT to a convolution and then using a fast convolution algorithm [58, 97, 68]. In this case no elementary FFTs are needed, being replaced by another algorithm.

1.9 Choice of FFT algorithms

Athough, as mentioned above, the choice between the FFT algorithms is highly machine-dependent, we can state that the prime factor algorithm uses fewer complex arithmetic operations than the Cooley-Tukey algorithm. Replacing a prime factor-based computation in a schema with a Cooley-Tukey computation will therefore increase the number of operations required.

It is better to perform the Cooley-Tukey algorithm first, and prime factor-based computations later. Consider computing a FFT of size $p^a q^a$, where p and q are relatively prime. If we compute a radix-pq Cooley-Tukey algorithm, then use the prime factor algorithm to reduce the FFTs of size pq to size p and q, we will perform $a - 1$ twiddle factor multiplications. If instead we use the prime factor algorithm to reduce FFT to FFTs of size p^a and q^a, then use Cooley-Tukey to compute these, we will perform $2(a - 1)$ twiddle factor multiplications, and otherwise do exactly the same amount of work.

The Rader algorithm is generally more efficient in terms of floating-point operations than Bluestein, primarily for these reasons:

- In the Rader method, it may sometimes be possible to compute FFTs of size $n - 1$. This may produce a factor-of-two saving, or even more; for instance, computing an FFT of order 7 using Rader's method requires two sub-FFTs of order 6; using Bluestein, the sub-FFTs must be of order at least 15.

- Bluestein's method requires two componentwise multiplications on the input and output vectors; Rader's method does not.

There are some cases, even at this level of abstraction, when the choice is not clear-cut; for instance, the choice between using Bluestein's algorithm or moving directly to a Cooley-Tukey framework. The problem becomes even more complex when the cost of data movement is considered.

[14] attempted a comparison of prime-factor and Cooley-Tukey based FFT variations, and the subsequent correspondence included [19]. [85] is a similar comparison for vector computers which indicated that "prime factor algorithms offer only very modest improvements over the conventional forms of the algorithm".

1.10 Data Movement in Cooley-Tukey FFT Variants

If we are interested in an FFT framework that can handle many (or all) values of n, it will be helpful to use a Cooley-Tukey-based FFT at some stage. Although the Cooley-Tukey FFTs all use the same underlying mathematics (equation 1.29), they vary in the data ordering and in the factorization of the FFT length that is chosen at each recursive step.

To introduce the notation we will be using, consider the original Cooley-Tukey algorithm as presented in [24]. This algorithm, for the radix-2 case applied to an FFT of length 2^5, is represented in diagrammatic form in figure 1.1. The diagram shows the operation of the algorithm on the indices of the vector.

The first step of the algorithm is an order-2 DFT, performed on the most significant bit; if the initial vector is $X(j_0, j_1, j_2, j_3, j_4)$ where j_0 is the most significant index and the various j_i can have values 0 or 1, then the first step calculates an order-2 DFT on only the most significant array index. An DFT of order 2 is particularly simple; to compute $y = F_2 x$, one calculates

$$(y)_0 := (x)_0 + (x)_1 \tag{1.54}$$

$$(y)_1 := (x)_0 - (x)_1 \tag{1.55}$$

The second step, labelled "scaling", consists of the componentwise multiplication by powers of ω, the "twiddle factor" from equation 1.29. The third step is another order-2 DFT. The fourth step is again a twiddle factor multiplication, and so on. Finally, there is a bit reversal.

This same computation is presented below in a different, vector-based notation, in which algorithms are presented as sequences of linear transformations on some vector X. When we write that 'X is indexed by q, r', we mean that the length of X is qr, and that $X(j, k)$ is the $(1 + j + kr)$th element of X; we will usually consider q and r to be powers of 2, in which case indices to the right will supply less significant bits of the final index (we are here trying to be consistent with the way numbers are written in the usual Arabic notation). The notation $Y(j, \cdot) := TX(j, \cdot)$ means that the linear transformation T operates over the second index of X to produce Y.

In this notation, the original Cooley-Tukey algorithm for the case $n = 2^5$, using radix 2, looks like this: If X is indexed by $2, 2, 2, 2, 2$,

$$Z^{(1)}(\cdot, j_3, j_2, j_1, j_0) := F_2 X(\cdot, j_3, j_2, j_1, j_0) \tag{1.56}$$

$$Z^{(2)}(j_4, j_3, j_2, j_1, j_0) := \omega_{2^2}^{j_4 j_3} Z^{(1)}(j_4, j_3, j_2, j_1, j_0) \tag{1.57}$$

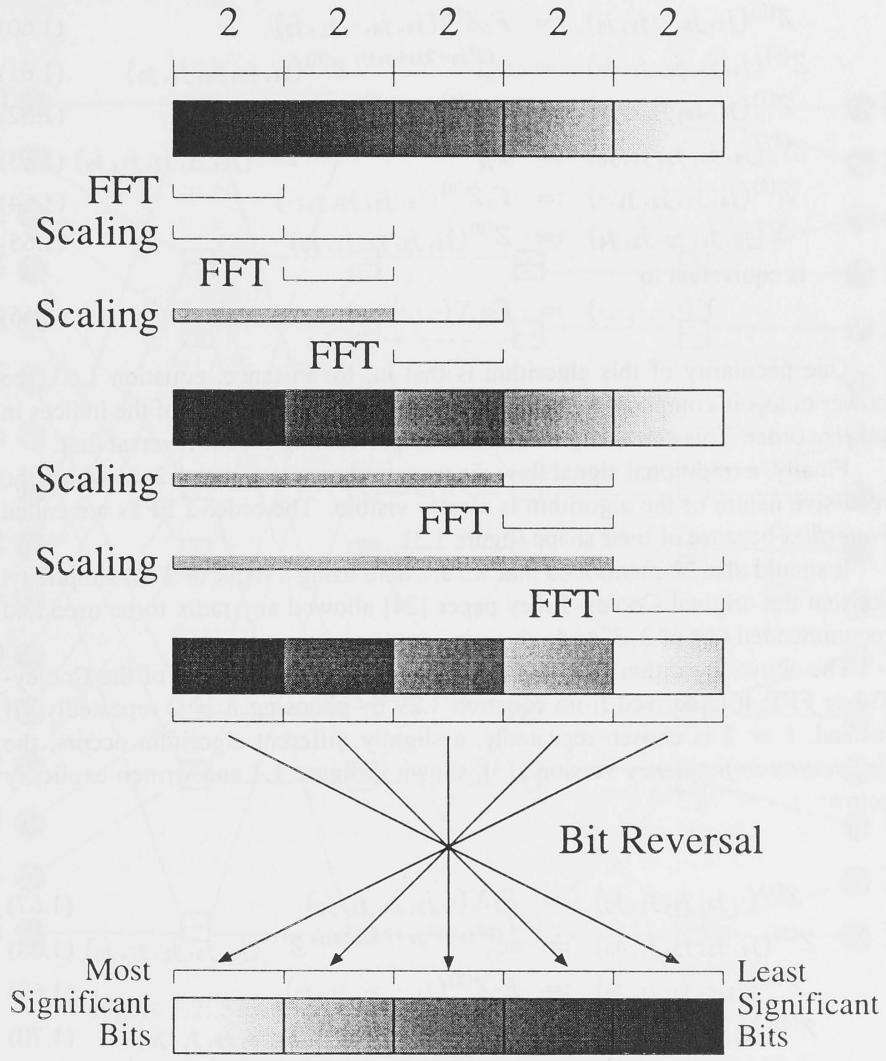


Figure 1.1: The original Cooley-Tukey FFT as presented in [24], size 2^5, radix 2.

$$Z^{(3)}(j_4,\cdot,j_2,j_1,j_0) := F_2 Z^{(2)}(j_4,\cdot,j_2,j_1,j_0) \quad (1.58)$$

$$Z^{(4)}(j_4,j_3,j_2,j_1,j_0) := \omega_{2^3}^{(2j_3+j_4)j_2} Z^{(3)}(j_4,j_3,j_2,j_1,j_0) \quad (1.59)$$

$$Z^{(5)}(j_4,j_3,\cdot,j_1,j_0) := F_2 Z^{(4)}(j_4,j_3,\cdot,j_1,j_0) \quad (1.60)$$

$$Z^{(6)}(j_4,j_3,j_2,j_1,j_0) := \omega_{2^4}^{(2^2j_2+2j_3+j_4)j_1} Z^{(5)}(j_4,j_3,j_2,j_1,j_0) \quad (1.61)$$

$$Z^{(7)}(j_4,j_3,j_2,\cdot,j_0) := F_2 Z^{(6)}(j_4,j_3,j_2,\cdot,j_0) \quad (1.62)$$

$$Z^{(8)}(j_4,j_3,j_2,j_1,j_0) := \omega_{2^5}^{(2^3j_1+2^2j_2+2j_3+j_4)j_0} Z^{(7)}(j_4,j_3,j_2,j_1,j_0) \quad (1.63)$$

$$Z^{(9)}(j_4,j_3,j_2,j_1,\cdot) := F_2 Z^{(8)}(j_4,j_3,j_2,j_1,\cdot) \quad (1.64)$$

$$Y(j_0,j_1,j_2,j_3,j_4) := Z^{(9)}(j_4,j_3,j_2,j_1,j_0) \quad (1.65)$$

is equivalent to

$$Y(\cdot,\cdot,\cdot,\cdot,\cdot) := F_{2^5} X(\cdot,\cdot,\cdot,\cdot,\cdot) \quad (1.66)$$

One pecularity of this algorithm is that in, for instance, equation 1.63, the power of ω_{2^5} is computed by using the most significant four bits of the indices in *reverse* order. This can easily be avoided by performing the bit-reversal first.

Finally, a traditional signal flow diagram is shown in figure 1.2, in which the recursive nature of the algorithm is clearly visible. The order-2 FFTs are called *butterflies* because of their shape (figure 1.3).

It should also be mentioned that we are here using a radix of 2 for simplicity, but that the original Cooley-Tukey paper [24] allowed any radix to be used and recommended one of 2, 3, or 4.

The above algorithm is called the *decimation in time* variant of the Cooley-Tukey FFT; it is derived from equation 1.29 by choosing $r = 2$ repeatedly. If instead, $t = 2$ is chosen repeatedly, a slightly different algorithm occurs, the *decimation in frequency* version [13], shown in figure 1.4 and written explicitly below:

$$Z^{(1)}(\cdot,j_3,j_2,j_1,j_0) := F_2 X(\cdot,j_3,j_2,j_1,j_0) \quad (1.67)$$

$$Z^{(2)}(j_4,j_3,j_2,j_1,j_0) := \omega_{2^5}^{(2^3j_3+2^2j_2+2j_1+j_0)j_4} Z^{(1)}(j_4,j_3,j_2,j_1,j_0) \quad (1.68)$$

$$Z^{(3)}(j_4,\cdot,j_2,j_1,j_0) := F_2 Z^{(2)}(j_4,\cdot,j_2,j_1,j_0) \quad (1.69)$$

$$Z^{(4)}(j_4,j_3,j_2,j_1,j_0) := \omega_{2^4}^{(2^2j_2+2j_1+j_0)j_3} Z^{(3)}(j_4,j_3,j_2,j_1,j_0) \quad (1.70)$$

$$Z^{(5)}(j_4,j_3,\cdot,j_1,j_0) := F_2 Z^{(4)}(j_4,j_3,\cdot,j_1,j_0) \quad (1.71)$$

$$Z^{(6)}(j_4,j_3,j_2,j_1,j_0) := \omega_{2^3}^{(2j_1+j_0)j_3} Z^{(5)}(j_4,j_3,j_2,j_1,j_0) \quad (1.72)$$

$$Z^{(7)}(j_4,j_3,j_2,\cdot,j_0) := F_2 Z^{(6)}(j_4,j_3,j_2,\cdot,j_0) \quad (1.73)$$

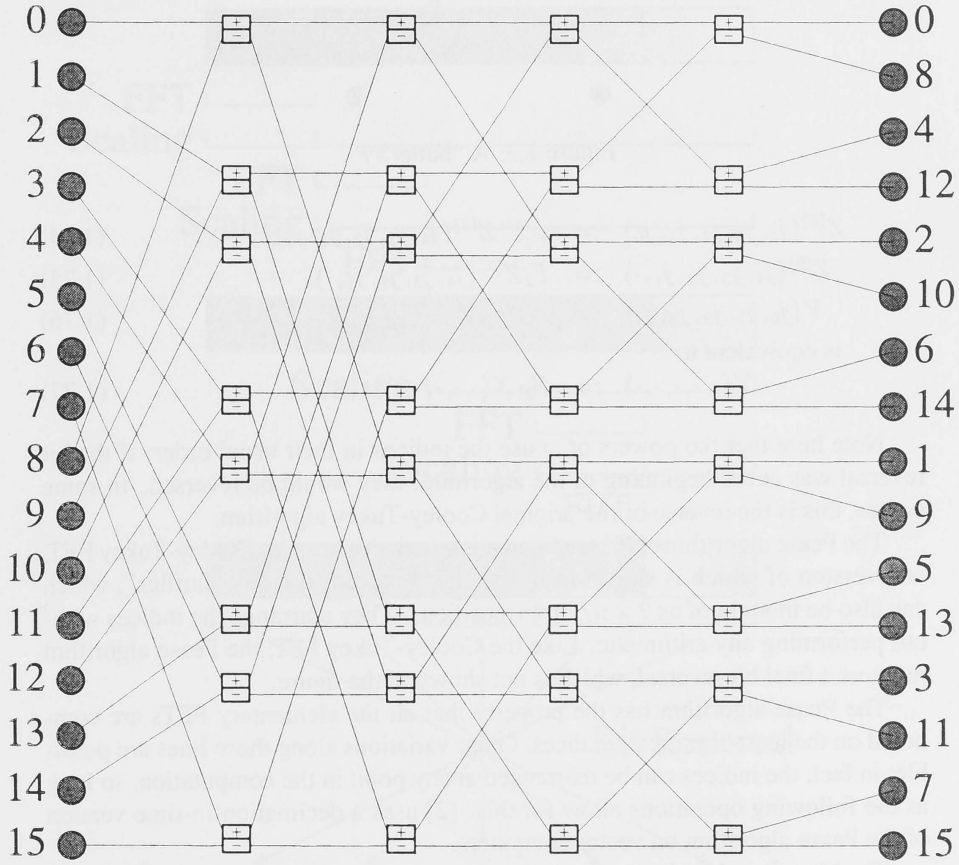


Figure 1.2: Signal flow for a size 2^4 radix 2 Cooley-Tukey FFT.

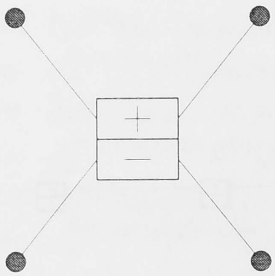


Figure 1.3: A "butterfly".

$$Z^{(8)}(j_4, j_3, j_2, j_1, j_0) := \omega_{2^2}^{j_0 j_3} Z^{(7)}(j_4, j_3, j_2, j_1, j_0) \tag{1.74}$$

$$Z^{(9)}(j_4, j_3, j_2, j_1, \cdot) := F_2 Z^{(8)}(j_4, j_3, j_2, j_1, \cdot) \tag{1.75}$$

$$Y(j_0, j_1, j_2, j_3, j_4) := Z^{(9)}(j_4, j_3, j_2, j_1, j_0) \tag{1.76}$$

is equivalent to

$$Y(\cdot, \cdot, \cdot, \cdot, \cdot) := F_{2^5} X(\cdot, \cdot, \cdot, \cdot, \cdot) \tag{1.77}$$

Note here that the powers of ω use the indices in their usual order; if the bit reversal was at the beginning of the algorithm, they would be reversed. In some senses, this is the inverse of the original Cooley-Tukey algorithm.

The Pease algorithms [70] are interesting variations on the Cooley-Tukey FFT, one version of which is shown in figure 1.5. Note the "perfect shuffles", which can also be thought of as $2 \times n/2$ transpositions. They rearrange the indices without performing any arithmetic. Like the Cooley-Tukey FFT, the Pease algorithm requires a final bit reversal, which is not shown in the figure.

The Pease algorithm has the property that all the elementary FFTs are computed on the least-significant indices. Other variations along these lines are possible; in fact, the indices can be rearranged at any point in the computation, so long as the following operations allow for this. [2] uses a decimation-in-time version of the Pease algorithm on vector computers.

The bit reversal in the previous algorithms has been examined at some length in its own right, both for sequential machines [53], and for parallel machines, particularly hypercubes [21, 92]. Of course, the details of the memory hierarchy on a machine are very significant when choosing a bit-reversal strategy [76]. The bit reversal can be quite expensive, particularly as it is not here overlapped with any arithmetic operations.

To estimate the quantity of communication required by an algorithm, one use-

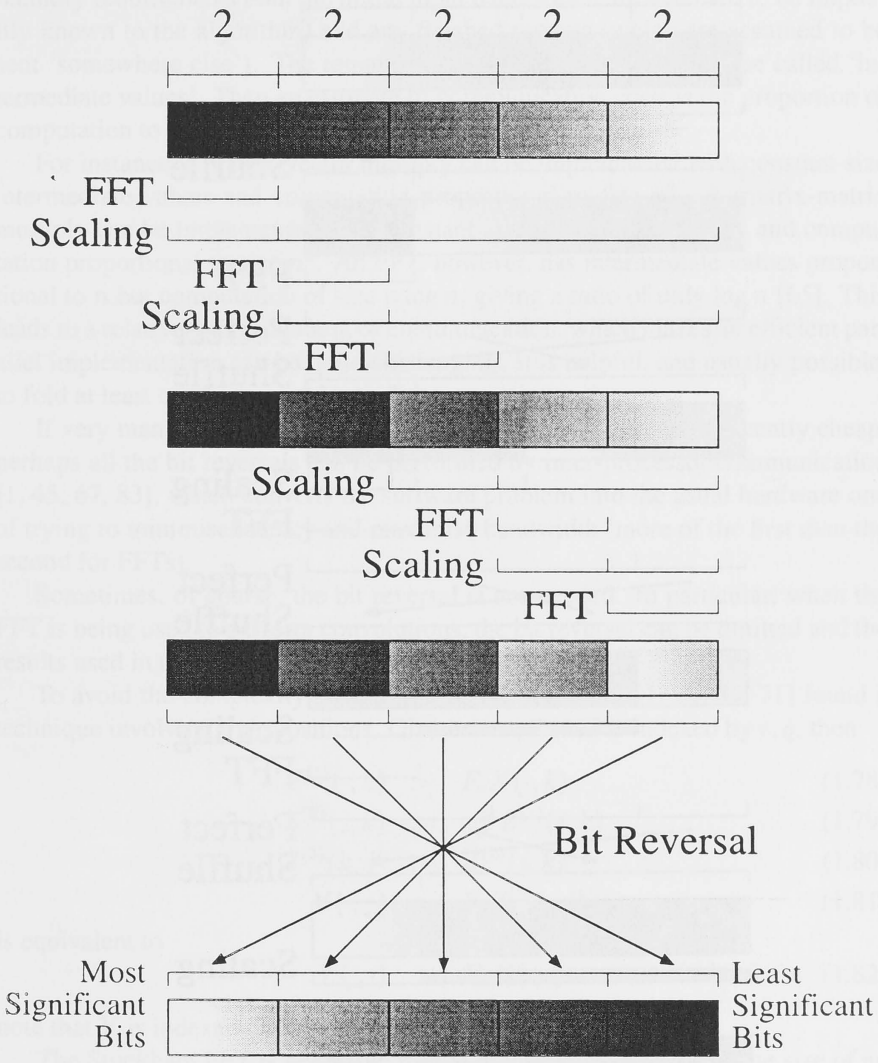


Figure 1.4: A decimation-in-frequency radix 2 Cooley-Tukey FFT of size 2^5.

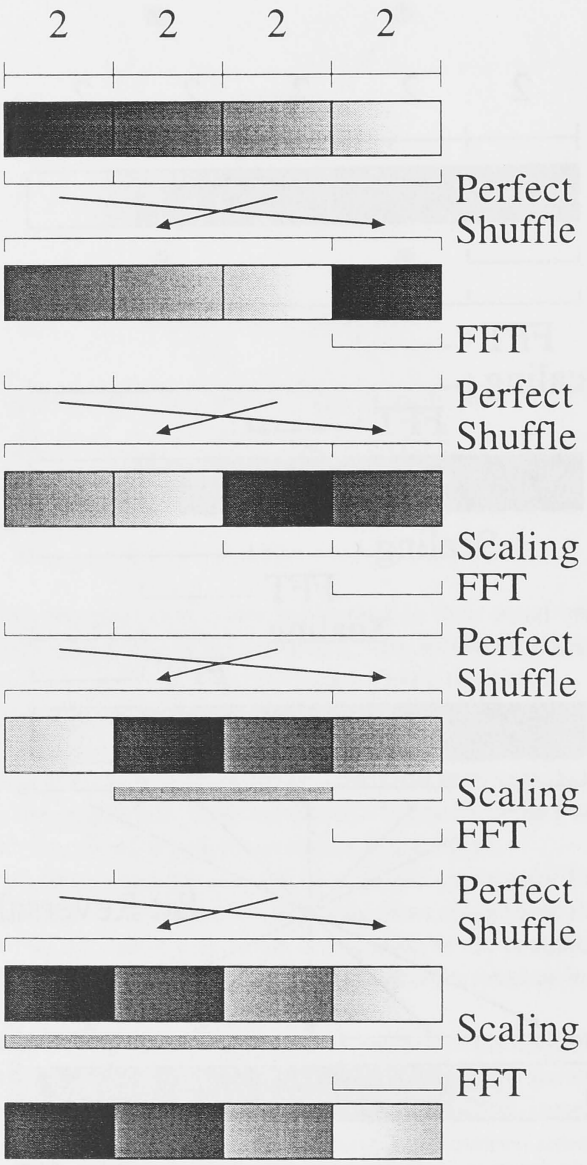


Figure 1.5: A Pease FFT [70] of size 2^4, radix 2.

ful statistic is the size of the intermediate values—that is, we exclude from the memory requirements both the initial input data (which are assumed to be implicitly known to the algorithm) and any finished outputs (which are assumed to be sent 'somewhere else'). The remaining memory requirements can be called 'intermediate values'. Then an estimate of communication costs is the proportion of computation to the size of the intermediate values.

For instance, a matrix-vector multiply can be implemented with constant-size intermediate values and computation proportional to size n^2. A matrix-matrix multiply can be implemented with constant-size intermediate values and computation proportional to size n^3. An FFT, however, has intermediate values proportional to n but computation of size $n \log n$, giving a ratio of only $\log n$ [65]. This leads to a relatively large amount of communication, which makes an efficient parallel implementation can be quite challenging. It is helpful, and usually possible, to fold at least one bit reversal into the communication.

If very many processors are available, or communication is sufficently cheap, perhaps all the bit reversals can be performed by inter-processor communication [1, 45, 67, 83], which converts the software problem into the usual hardware one of trying to minimise latency and maximise bandwidth (more of the first than the second for FFTs).

Sometimes, of course, the bit reversal is not needed. In particular, when the FFT is being used to perform convolutions, the bit reversal can be omitted and the results used in the bit-reversed order [86].

To avoid the complexity of a bit reversal, Stockham [81, 22, 79, 71] found a technique involving transpositions. Given $n = qr$, if X is indexed by r, q, then

$$Z^{(1)}(\cdot, k) := F_r X(\cdot, k) \tag{1.78}$$

$$Z^{(2)}(j, k) := \omega_n^{jk} Z^{(1)}(j, k) \tag{1.79}$$

$$Z^{(3)}(k, j) := Z^{(2)}(j, k) \tag{1.80}$$

$$Y(\cdot, j) := F_q Z^{(3)}(\cdot, j) \tag{1.81}$$

is equivalent to

$$Y(\cdot, \cdot) := F_n X(\cdot, \cdot). \tag{1.82}$$

note that Y is indexed by q, r.

The Stockham FFT is generated by repeatedly choosing q to be the size of an elementary FFT (for instance, 2); the transposed Stockham FFT is generated by choosing r to be the size of an elementary FFT.

The *four-step algorithm* is the general algorithm described above [83], used with q and r approximately equal, which allows long vector lengths on vector

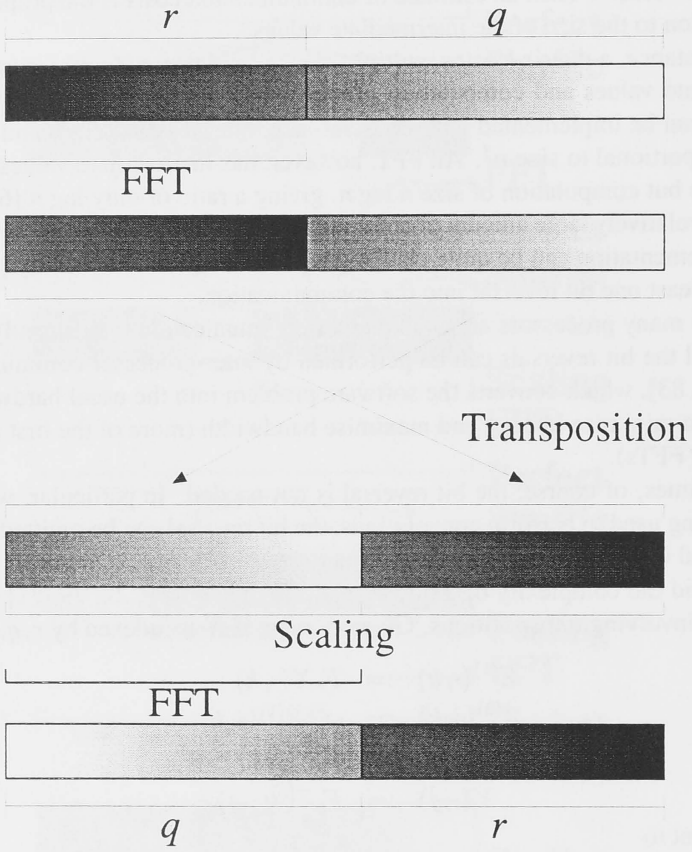


Figure 1.6: The basic step in a Stockham or four-step FFT.

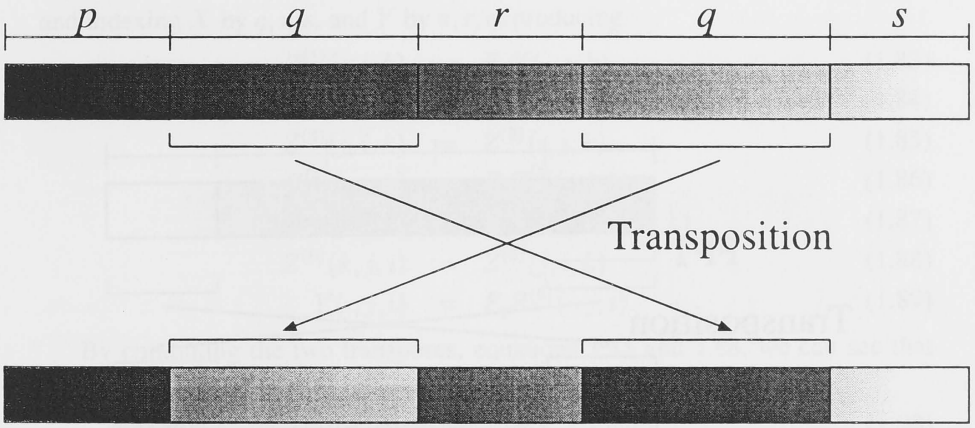


Figure 1.7: A "square transposition".

processors.

This algorithm does ensure that no bit reversal is necessary—this is the *self-sorting* property. However, it is difficult to code the transposition operation in the four-step algorithm without using an extra workspace of size n; this is unfortunate, because it can double the workspace requirements of the FFT computation compared to a Cooley-Tukey algorithm without the final bit reversal. An algorithm with this property—that it can compute its result without needing more than constant-sized workspace—is called *in-place*.

The essential tool for building practical in-place FFT algorithms is the following observation: although it is quite difficult to perform an arbitrary transposition of two indices (such as in figure 1.6) in-place, if the indices have equal range (as shown in figure 1.7), then the transposition can be performed in-place. We will call this a *square transposition*.

We will present two algorithms for computing in-place self-sorting FFTs. The first is due to Johnson and Burrus [47] in the radix-2 case, and Temperton [89] in the general mixed-radix case, and a radix-2 version of size 2^5 in figure 1.8. A distinguishing feature of this algorithm is that all the transpositions occur in the first half of the computation; after that, only FFTs and scalings are performed.

To produce an algorithm suitable for vector processing, [39, 40, 56] the four-step algorithm can be applied to itself, by factoring the size of the FFT $n = q \cdot (rs)$

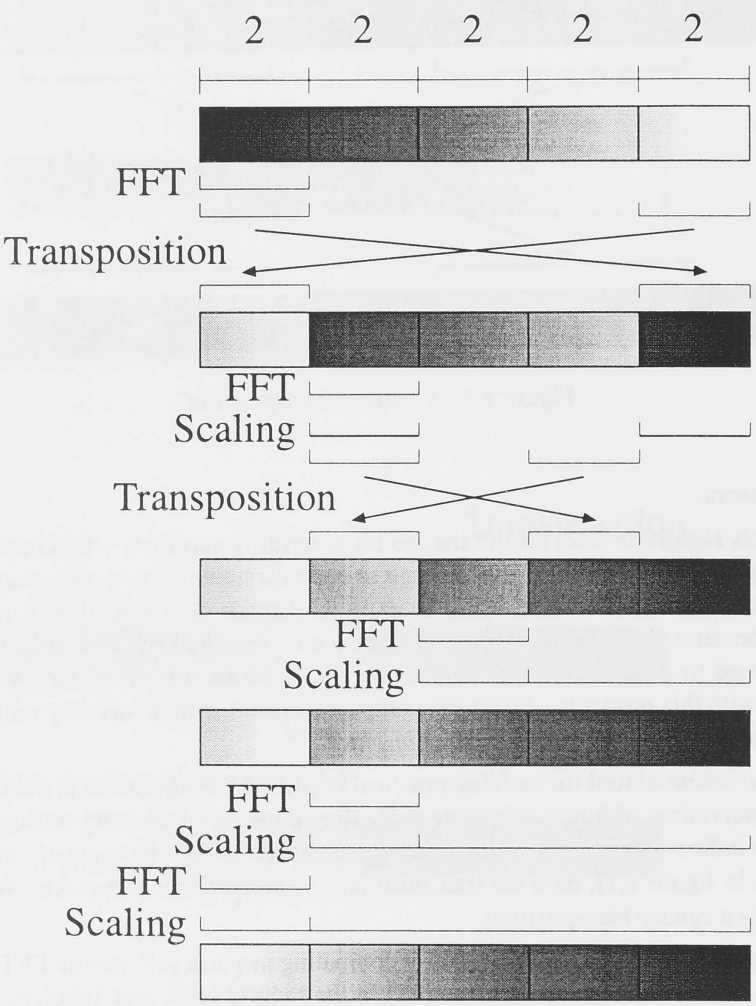


Figure 1.8: The Johnson and Burrus [47] in-place self-sorting FFT of size 2^5.

and indexing X by q, r, s, and Y by s, r, q, producing

$$Z^{(1)}(\cdot, j, k) := F_s X(\cdot, j, k) \quad (1.83)$$
$$Z^{(2)}(i, j, k) := \omega_{rs}^{ij} Z^{(1)}(i, j, k) \quad (1.84)$$
$$Z^{(3)}(j, i, k) := Z^{(2)}(i, j, k) \quad (1.85)$$
$$Z^{(4)}(\cdot, i, k) := F_r Z^{(3)}(\cdot, i, k) \quad (1.86)$$
$$Z^{(5)}(j, i, k) := \omega_n^{k(sj+i)} Z^{(4)}(j, i, k) \quad (1.87)$$
$$Z^{(6)}(k, j, i) := Z^{(5)}(j, i, k) \quad (1.88)$$
$$Y(\cdot, j, i) := F_q Z^{(6)}(\cdot, j, i). \quad (1.89)$$

By combining the two transposes, equations 1.85 and 1.88, we can see that this is equivalent to

$$Z^{(1)}(\cdot, j, k) := F_s X(\cdot, j, k) \quad (1.90)$$
$$Z^{(2)}(i, j, k) := \omega_{rs}^{ij} Z^{(1)}(i, j, k) \quad (1.91)$$
$$Z^{(3)}(k, j, i) := Z^{(2)}(i, j, k) \quad (1.92)$$
$$Z^{(4)}(k, \cdot, i) := F_r Z^{(3)}(k, \cdot, i) \quad (1.93)$$
$$Z^{(5)}(k, j, i) := \omega_n^{k(sj+i)} Z^{(4)}(k, j, i) \quad (1.94)$$
$$Y(\cdot, j, i) := F_q Z^{(5)}(\cdot, j, i). \quad (1.95)$$

The index behaviour of this algorithm is shown in figure 1.9. Usually, r is chosen to be a size that can be computed by an elementary FFT, and to have an in-place algorithm q and s must be chosen so that $q = s$. Much more will be said about this algorithm, and its implementation, in a subsequent chapter.

1.11 Prime Factor Algorithm Implementation

It is possible to implement the prime factor algorithm in-order and in-place using appropriate elementary FFT routines. The initial description of this algorithm was done in [20], with improvements in [75] and [87, 88]. The primary difficulty to be overcome is the complicated indexing the prime factor algorithm uses.

1.12 Accuracy

In general, using the FFT is more accurate than applying the DFT as a matrix-vector product; the round-off error is reduced by a factor of $\frac{n}{\log n}$, as each result depends on fewer computations [22].

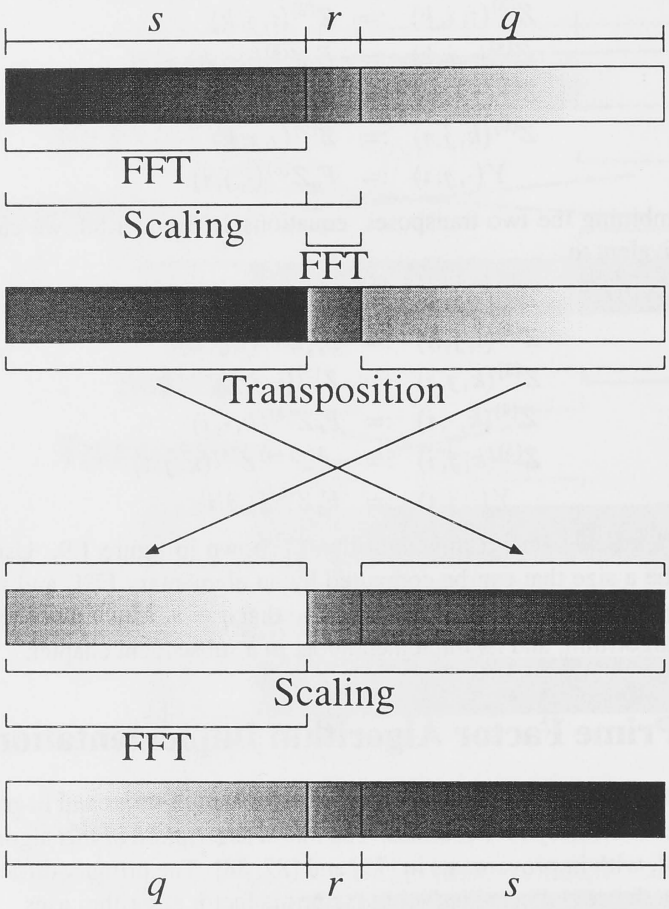


Figure 1.9: The six-step FFT algorithm.

In [4], Arioli and others analysed the computation of a radix-2 Cooley-Tukey FFT of size n, assuming that the twiddle factors are computed exactly and then rounded. The results were that, if x is the input vector, Fx is the true result, and $fl(Fx)$ is the computed result, then the componentwise relative error is bounded by, in the worst case,

$$\frac{||fl(Fx) - Fx||_\infty}{||Fx||_\infty} < 10.7\sqrt{n}\epsilon + O(\epsilon^2) \tag{1.96}$$

where ϵ is the machine precision. The distance (the L_2 norm) between the true result and the computed value is, in the worst case,

$$\frac{||fl(Fx) - Fx||_2}{||Fx||_2} < 10.7 \cdot \log_{10}(n) \cdot \epsilon + O(\epsilon^2). \tag{1.97}$$

Under some apparently reasonable assumptions about the statistical distribution of the errors, Arioli found the expected maximum componentwise relative error is bounded by

$$E\left[\frac{||fl(Fx) - Fx||_\infty}{||Fx||_\infty}\right] \leq \epsilon\sqrt{\frac{2}{\ln 2}}\ln n + O(\epsilon) \tag{1.98}$$

Recently, some interesting work [29] was done on trading accuracy of the FFT for reduced communication cost using a fast multipole technique, so that most of the the accuracy that is lost would not have been achieved anyway due to round-off error. The technique is particularly useful on multiprocessor systems. In [38] a similar technique using wavelets is presented, which has the added advantage that the approximation technique can be used to reduce any noise in the input signal.

1.13 Application of the FFT: Integer Multiplication

One application of the FFT, which is presented here because of its significance, is in the multiplication of integers. Suppose we wish to multiply two n-digit integers, t and u. The traditional method is to write t and u as vectors x and y of digits:

$$t = \sum_{j=0}^{n-1} 10^j (x)_j \tag{1.99}$$

$$u = \sum_{j=0}^{n-1} 10^j (y)_j \tag{1.100}$$

```
                4 8 5 1 6 ×
                5 1 9 5 4
              -------------
            1 9 4 0 6 4     This is 48516x4,
          2 4 2 5 8 0       this is 48516x5,
        4 3 6 6 4 4         and so on...
        4 8 5 1 6
2 4 2 5 8 0
-------------------------
2 5 2 0 6 0 0 2 6 4         This is the sum of the above
                            values aligned as shown.
```

Figure 1.10: Multiplication form.

Then to perform the multiplication $v = tu$, we compute a vector z of length $2n$:

$$(z)_k := \sum_{j=\max(0,k-n-1)}^{\min(n-1,k)} (x)_j (y)_{k-j} \tag{1.101}$$

$$v := \sum_{i=0}^{2n-1} 10^i (z)_i \tag{1.102}$$

$$\tag{1.103}$$

Schoolchildren are taught to use this method using a form similar to that in figure 1.10. This method requires n^2 single-digit multiplications for equation 1.101. Of course, before producing a final result it is necessary to reduce any overlarge elements of z down to the range of a digit.

However, equation 1.101 is similar in form to equation 1.1. Suppose we extend vectors x and y to length $2n$ by appending zeroes so that equations 1.99 and 1.100 still hold. Then whenever $0 \leq j < k - n - 1$, $(y)_{k-j} = 0$ (because $k - j$ will be between $-n$ and -1); and whenever $n - 1 < j < 2n$, $(x)_j = 0$. So we can rewrite equation 1.101 as:

$$(z)_k = \sum_{j=0}^{2n-1} (x)_j (y)_{k-j} \tag{1.104}$$

This is a convolution; and we can use the convolution property to compute this with $O(n \log n)$ operations.

In the above, observe that the number 10 can be replaced with any base β, including negative and complex 'bases' [57]. In fact, we can substitute an unbound variable x instead of the base and use this technique to multiply polynomials.

Chapter 2

The Multidimensional Fourier Transform

In this chapter, a more general way of deriving the DFT will be presented. This will allow the structures described in the previous chapter to be applied to more general situations, in particular the case of multidimensional FFTs. It will also allow the easy derivation of the DFT in the one-dimensional case; this is the proof that was skipped over in section 1.2.2.

The basic group-theoretic definitions we will use are provided in **Appendix B**.

The approach taken here will be aimed towards DFTs on finite groups, and on the cyclic groups in particular. For a treatment of DFTs on infinite groups (compact Lie groups in particular) and arbitrary finite groups, see [62]. An alternative notation, less formal than the group-theoretic one used here, is that of index-digit permutations, see for instance [31].

For a historical view of convolutions on multidimensional data, see [63] and the papers it references.

2.1 Generalised Convolutions

We define the generalised convolution operation following [3]. Let G be a finite group written additively. Then the *group ring* of G over the complex numbers, written $\mathbb{C}G$, is the algebra $(\mathbb{C}G, +, \cdot, \star)$ consisting of all vectors over $\mathbb{C}$ indexed by elements of G, with the operations defined as follows: if $a, b \in \mathbb{C}G$, $\alpha \in \mathbb{C}$,

then for all $g \in G$,

$$(a+b)_g = a_g + b_g \tag{2.1}$$

$$(\alpha \cdot a)_g = \alpha a_g \tag{2.2}$$

$$(a \star b)_g = \sum_{h \in G} a_h b_{g-h} \tag{2.3}$$

$\star$ will be recognised as the same operation described by equation 1.1 in the previous chapter.

This approach can be extended to infinite groups by treating members of $\mathbb{C}G$ as functions from G to $\mathbb{C}$ and using an appropriate measure instead of the summation in equation 2.3.

For ease of notation, we will introduce a vector space basis consisting of the vectors v_g, so that for all $g, h \in G$:

$$(v_g)_h = \begin{cases} 1, & \text{if } g = h; \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

There is a group homomorphism $f : G \to (\mathbb{C}G, \star)$ defined by $f(g) = v_g$; in particular, $v_{g+h} = v_g \star v_h$. Any $a \in \mathbb{C}G$ can be written as $a = \sum_{g \in G} a_g \cdot v_g$.

With this multiplication and addition, $(\mathbb{C}G, +, \star)$ is a ring with 1 ($1_{\mathbb{C}G} = v_0$), and as $\star$ is $\mathbb{C}$-linear, $\mathbb{C}G$ is a $\mathbb{C}$-algebra.

2.2 The Generalised Discrete Fourier Transform

The next theorem, called Maschke's Theorem, is the heart of the generalised DFT. The proofs here are from [3].

2.2.1 Modules

Theorem 2 *Let G be a group, $\mathbb{F}$ be a field, and suppose that the characteristic of $\mathbb{F}$ is either zero or coprime to $|G|$. If U is an $\mathbb{F}G$-module and S is an $\mathbb{F}G$-submodule of U, then there exists some T, an $\mathbb{F}G$-submodule of U, so that $U = S \oplus T$.*

Consider S as an $\mathbb{F}$-vector subspace of U. Then there is some subspace W of U so that $U = S \oplus W$ as $\mathbb{F}$-vector spaces; however, W may not be an $\mathbb{F}G$-submodule of U.

Let $\pi : U \to S$ be the projection of U onto S along W, so that π is the unique $\mathbb{F}$-linear transformation that is the identity on S and zero on W. We define an $\mathbb{F}$-linear transformation $\pi' : U \to S$ by

$$\pi'(u) = (|G|1_{\mathbb{F}})^{-1} \cdot \sum_{g \in G} v_g \pi(v_{-g} u) \tag{2.5}$$

This requires that $|G|1_{\mathbb{F}} \neq 0_{\mathbb{F}}$, which is equivalent to the hypothesis about the characteristic of $\mathbb{F}$.

Since S is a submodule of U, then for any $g \in G$ and $s \in S$, $v_g s \in S$ and so the image of π' lies in S. Since π is the identity on S, then

$$\pi'(s) = (|G|1_{\mathbb{F}})^{-1} \sum_{g \in G} v_g v_{-g} s = s \tag{2.6}$$

and so π' restricted to S is also the identity. It now follows from linear algebra that $U = S \oplus \ker \pi'$ as $\mathbb{F}$-vector spaces.

To show that $\ker \pi'$ is an $\mathbb{F}G$-submodule of U, it suffices to show that π' is an $\mathbb{F}G$-module homomorphism; to show that, it is sufficient that for any $x \in G$, $u \in U$, $\pi'(v_x u) = v_x \pi'(u)$ because we already know that π' is $\mathbb{F}$-linear. So,

$$\begin{aligned}
\pi'(v_x u) &= (|G|1_{\mathbb{F}})^{-1} \sum_{g \in G} v_g \pi(v_{-g} v_x u) && (2.7)\\
&= (|G|1_{\mathbb{F}})^{-1} \sum_{g \in G} v_x v_{-x} v_g \pi(v_{-g} v_x u) && (2.8)\\
&= v_x (|G|1_{\mathbb{F}})^{-1} \sum_{g \in G} v_{g-x} \pi(v_{x-g} u) && (2.9)
\end{aligned}$$

Substitute $y = g - x$:

$$\begin{aligned}
&= v_x (|G|1_{\mathbb{F}})^{-1} \sum_{y \in G} v_y \pi(v_{-y} u) && (2.10)\\
&= v_x \pi'(u) && (2.11)
\end{aligned}$$

as required. □

A module is said to be *simple* if it has no submodules other than itself and the zero module (the zero module is not usually considered simple).

Theorem 3 *Let G be a group, $\mathbb{F}$ be a field, and suppose that the characteristic of $\mathbb{F}$ is either zero or coprime to $|G|$. If U is a finite-dimensional non-zero $\mathbb{F}G$-module then U can be written as direct sums of simple submodules.*

The proof will be by induction on the dimension of U as an $\mathbb{F}$-vector space. If the dimension of U is 1, then U must be simple. If U is simple, then the theorem is trivially true.

So now suppose that the dimension of U is greater than 1, and U has a proper non-zero $\mathbb{F}G$-submodule V. Then by Maschke's Theorem, $U = V \oplus W$ for some proper non-zero submodule W of U. But both V and W must have dimension strictly less than that of U, and so by induction they can be written as direct sums of simple modules; so U can also be written as such a direct sum. □

2.2.2 Algebras

The preceeding theorem implies that all $\mathbb{F}G$-modules are semisimple. Algebras with this property are also called *semisimple*. It turns out that if we define a *simple* algebra to be one which has no nontrivial ideals, then we can write each semisimple algebra as a direct sum of simple algebras.

To be able to write this explicitly, we need to define a way to link A as an A-module with A the algebra. Suppose M is an A-module. Then define $\mathrm{End}_A(M)$ to be the algebra of A-module homomorphisms from M to M, in which the ring operation is defined by composition so that if $\phi, \psi \in \mathrm{End}_A(M)$ and $m \in M$, then $(\phi \star \psi)(m) = \psi(\phi(m))$. Note that the ring operation here is the reverse of what might be expected.

The important property of this is that, as algebras,

$$A = \mathrm{End}_A(A). \tag{2.12}$$

This allows us to write the following theorem, which is named Wedderburn's structure theorem for algebras. Note here that $n_i S_i$ means $S_i \oplus S_i \oplus \ldots \oplus S_i$ where there are n_i summands.

Theorem 4 *Suppose A is a semisimple algebra. Then we can write, as A-modules, $A = n_1 S_1 \oplus n_2 S_2 \oplus \ldots \oplus n_m S_m$, where S_i are distinct simple A-modules, and n_i are integers. Furthermore, as algebras,*

$$A = \mathcal{M}_{n_1}(\mathrm{End}_A(S_1)) \oplus \mathcal{M}_{n_2}(\mathrm{End}_A(S_2)) \oplus \ldots \oplus \mathcal{M}_{n_m}(\mathrm{End}_A(S_m)).$$

This is theorem 16 in chapter 13 of [3]. □

Since $\mathbb{C}G$ is semisimple, then we can apply the preceeding theorem, and write $\mathbb{C}G$ as a direct sum. Then there is an algebra isomorphism F between $\mathbb{C}G$ and the direct sum; this isomorphism is the generalised discrete Fourier transform.

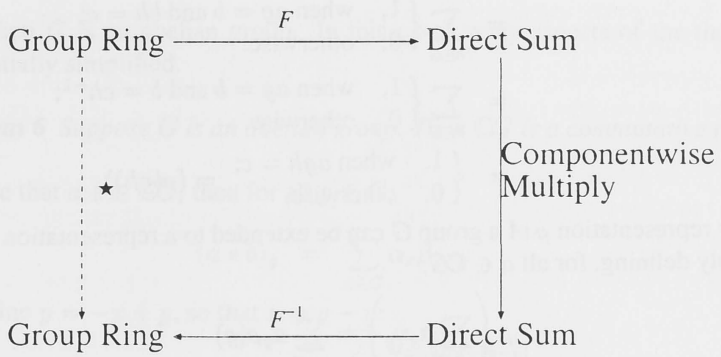


Figure 2.1: The generalised DFT can be used to compute generalised convolutions.

Figure 2.1 shows the practical use of this isomorphism that we are interested in: the product in the group ring can be computed by performing a generalised FFT, then the product in each component of the direct sum, then the inverse generalised FFT. This is simply a restatement of the fact that F is an isomorphism, and corresponds to equation 1.3 in the case where G is $\mathbb{Z}_n$ (see the next sections).

2.3 Representations

In the previous chapter, we referred to writing the DFT in matrix form. It is also possible to write a generalised DFT in matrix form.

A (matrix) *representation* of a group G is a homomorphism from G into the group of complex matrices (of some given size) under multiplication. Similarly, a representation of a $\mathbb{C}$-algebra A is an algebra homomorphism from A into the algebra of complex matrices of some given size.

The *regular representation* of a group G is a homomorphism ρ from G into the group of nonsingular $|G| \times |G|$ matrices indexed by elements of G, defined by:

$$(\rho(g))_{a,b} = \begin{cases} 1, & \text{when } ag = b; \\ 0, & \text{otherwise.} \end{cases} \tag{2.13}$$

We check that this is really a representation by computing

$$(\rho(g)\rho(h))_{a,c} = \sum_{b \in G} (\rho(g))_{a,b} (\rho(h))_{b,c} \tag{2.14}$$

$$= \sum_{b \in G} \begin{cases} 1, & \text{when } ag = b \text{ and } bh = c; \\ 0, & \text{otherwise.} \end{cases} \tag{2.15}$$

$$= \sum_{b \in G} \begin{cases} 1, & \text{when } ag = b \text{ and } b = ch^{-1}; \\ 0, & \text{otherwise.} \end{cases} \tag{2.16}$$

$$= \begin{cases} 1, & \text{when } agh = c; \\ 0, & \text{otherwise.} \end{cases} = (\rho(gh))_{a,c} \tag{2.17}$$

Any representation ρ of a group G can be extended to a representation of $\mathbb{C}G$, by simply defining, for all $a \in \mathbb{C}G$:

$$\rho\left(\sum_{g \in G} a_g v_g\right) = \sum_{g \in G} a_g \rho(g) \tag{2.18}$$

and conversely a representation of $\mathbb{C}G$ can be restricted to a representation of G.

The generalised convolution matrices are then the regular representation matrices of the elements of $\mathbb{C}G$. What of the result of the generalised DFT? We have the following information:

Theorem 5 *There are some positive integers r and $f_1, \ldots, f_r$ so that $\mathbb{C}G = \mathcal{M}_{f_1}(\mathbb{C}) \oplus \cdots \oplus \mathcal{M}_{f_r}(\mathbb{C})$ as $\mathbb{C}$-algebras. Furthermore, there are exactly r isomorphism classes of simple $\mathbb{C}G$-modules, and if we let $S_1, \ldots, S_r$ be representatives of these r classes, then we can order the S_i so that $\mathbb{C}G = f_1 S_1 \oplus \cdots \oplus f_r S_r$ as $\mathbb{C}G$-modules, where the dimension of S_i, as a $\mathbb{C}$-vector space, is f_i for each i.*

This is theorem 14.1 from [3]; its proof extends back over chapter 13 and part of chapter 12.

For the first part, this follows from the fact that if the field $\mathbb{F}$ is algebraically closed (as $\mathbb{C}$ is), then if A is an algebra over $\mathbb{F}$ and S is a simple A-module, then $\mathrm{End}_A(S) = \mathbb{F}$. □

Each column of $\mathcal{M}_{f_i}(\mathbb{C})$, treated as a vector, can be considered as a $\mathbb{C}G$-module where the module action is matrix-vector multiplication. With this action, each column is isomorphic to the simple module S_i.

From a computational point of view, this means that the result of the generalised DFT is best represented by the block-diagonal matrix formed by the $\mathcal{M}_{f_i}(\mathbb{C})$.

2.4 Abelian Groups

The determination of what the simple $\mathbb{C}G$-modules are for a particular G is beyond the scope of this document, but one particular case is important for us: the

case when G is an abelian group. In this case, many aspects of the theory are substantially simplified.

Theorem 6 *Suppose G is an abelian group. Then $\mathbb{C}G$ is a commutative ring.*

Suppose that $a, b \in \mathbb{C}G$, then for all $g \in G$,

$$(a \star b)_g = \sum_{x \in G} \alpha_x \beta_{g-x} \tag{2.19}$$

Define $y = -x + g$, so that $x = g - y$:

$$= \sum_{y \in G} \alpha_{g-y} \beta_{g-(g-y)} \tag{2.20}$$

$$= \sum_{y \in G} \alpha_{g-y} \beta_{g+y-g} \tag{2.21}$$

$$= \sum_{y \in G} \beta_y \alpha_{g-y} \tag{2.22}$$

$$= (b \star a)_g \tag{2.23}$$

as required. □

Theorem 7 *Suppose G is a finite abelian group. Then every simple $\mathbb{C}G$-module has dimension 1 as a $\mathbb{C}$-vector space, and $\mathbb{C}G = |G|\mathbb{C}$ as $\mathbb{C}$-algebras.*

Suppose there were some $n > 1$ and some $\mathbb{C}$-algebra A so that $\mathbb{C}G = \mathcal{M}_n(\mathbb{C}) \oplus A$ as $\mathbb{C}$-algebras. Then there is an algebra homomorphism ρ from $\mathbb{C}G$ onto $\mathcal{M}_n(\mathbb{C})$. Consider the matrices $M, N \in \mathcal{M}_n(\mathbb{C})$, which have every element zero except that $M_{1,0} = 1$ and $N_{0,0} = 1$. They have the property that $MN = M$ and $NM = 0$; that is, they do not commute. Since ρ is onto, there must be some $w, x \in \mathbb{C}G$ so that $\rho(w) = M$ and $\rho(x) = N$; but then

$$MN = \rho(w)\rho(x) = \rho(wx) = \rho(xw) = \rho(x)\rho(w) = NM$$

and so ρ cannot be an algebra homomorphism, therefore no such n can exist.

Thus, all the f_i in theorem 5 must be 1, which gives the first part of the proof directly and the second part follows from $\mathcal{M}_1(\mathbb{C}) = \mathbb{C}$ and the fact that two vector spaces can be isomorphic only if they have the same dimension. □

Now, what about the actual DFT? In section 1.2.2, we gave the result for the case when G is $\mathbb{Z}_n$, but did not prove its uniqueness. We can do that here, and generalise to the case of any finite abelian group. First, however, some notation.

The fundamental theorem on finite abelian groups is that every finite abelian group is a direct sum of cyclic groups. So in what follows, we will consider a finite abelian group $G = \mathbb{Z}_{n_0} \oplus \mathbb{Z}_{n_2} \oplus \ldots \oplus \mathbb{Z}_{n_{m-1}}$, written additively, with elements of G written as vectors indexed by 0 through $m-1$, so that $g_i \in \mathbb{Z}_{n_i}$. The group operation $+$ in G is then componentwise addition.

G is given a ring structure by considering each $\mathbb{Z}_{n_i}$ as a ring, then defining the product $\cdot$ on G as componentwise multiplication. Of course, the multiplication is also commutative.

Define a meaning for ω_G^g for $g \in G$:

$$\omega_G^g := \prod_{i=0}^{m-1} \omega_{n_i}^{g_i} \tag{2.24}$$

That is,

$$\omega_G^g = \exp\left(2\pi\sqrt{-1}\sum_{i=0}^{m-1}\frac{g_i}{n_i}\right) \tag{2.25}$$

ω_G has many of the usual properties of exponentials, in particular:

$$\omega_G^{a+b} = \omega_G^a \omega_G^b \tag{2.26}$$

$$\omega_G^{-a} = (\omega_G^a)^{-1} \tag{2.27}$$

Note that equation 2.26 is not quite as trivial as it appears; it relies on the fact that the function $f : \mathbb{Z}_{n_i} \to \mathbb{C}$ defined by $f(z) = \omega_{n_i}^z$ is a one-to-one group homomorphism.

Theorem 8 *Let G be a finite abelian group, $G = \mathbb{Z}_{n_0} \oplus \mathbb{Z}_{n_2} \oplus \ldots \oplus \mathbb{Z}_{n_{m-1}}$.*

Then each $k \in G$ gives rise to a simple $\mathbb{C}G$-module $(\mathbb{C}, +, \cdot_k)$ in which the action of $\mathbb{C}G$ on $\mathbb{C}$ is defined so that for all $a \in \mathbb{C}G$, $\alpha \in \mathbb{C}$,

$$a \cdot_k \alpha := \sum_{g \in G} a_g \omega_G^{kg} \alpha,$$

and $+$ is the usual complex addition.

Furthermore, if $k, l \in G$ and $k \neq l$, then the $\mathbb{C}G$-modules $(\mathbb{C}, +, \cdot_k)$ and $(\mathbb{C}, +, \cdot_l)$ are not isomorphic.

Finally, all simple $\mathbb{C}G$-modules are isomorphic to a module of the above form.

For the first part, we need to check that this does indeed define a module; in particular, that for all $a, b \in \mathbb{C}G$, $\alpha \in \mathbb{C}$, $(a \star b) \cdot_k \alpha = a \cdot_k (b \cdot_k \alpha)$:

$$a \cdot_k (b \cdot_k \alpha) = \left(\sum_{g \in G} a_g \omega_G^{kg} \right) \left(\sum_{h \in G} b_h \omega_G^{kh} \right) \alpha \tag{2.28}$$

$$= \sum_{g \in G} \sum_{h \in G} a_g b_h \omega_G^{k(g+h)} \alpha \tag{2.29}$$

Substitute $y = g + h$:

$$= \sum_{g \in G} \sum_{y \in G} a_g b_{y-g} \omega_G^{ky} \alpha \tag{2.30}$$

$$= \sum_{y \in G} \sum_{g \in G} a_g b_{y-g} \omega_G^{ky} \alpha \tag{2.31}$$

$$= \sum_{y \in G} (a \star b)_y \omega_G^{ky} \alpha \tag{2.32}$$

$$= (a \star b) \cdot_k \alpha \tag{2.33}$$

The module is 1-dimensional as a $\mathbb{C}$-vector space. Any proper submodule would have to have lesser dimension, that is dimension 0, so the submodule must be the trivial module, and so the module must be simple.

For the second part, suppose that there is a module isomorphism f from $(\mathbb{C}, +, \cdot_k)$ to $(\mathbb{C}, +, \cdot_l)$. Since it is a module isomorphism, then for any $g \in G$,

$$\omega_G^{lg} f(1_{\mathbb{C}}) = v_g \cdot_l f(1_{\mathbb{C}}) \tag{2.34}$$

$$= f(v_g \cdot_k 1_{\mathbb{C}}) \tag{2.35}$$

$$= f(\omega_G^{kg}) \tag{2.36}$$

$$= f((\omega_G^{kg} v_0) \cdot_k 1_{\mathbb{C}}) \tag{2.37}$$

$$= (\omega_G^{kg} v_0) \cdot_l f(1_{\mathbb{C}}) \tag{2.38}$$

$$= \omega_G^{kg} f(1_{\mathbb{C}}) \tag{2.39}$$

So, since $f(1_{\mathbb{C}}) \neq 0$, $\omega_G^{lg} = \omega_G^{kg}$. Then for each i, $0 \leq i < m$, define $u_i \in G$ as follows:

$$(u_i)_j := \begin{cases} 1, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases} \tag{2.40}$$

and then for each i choose $g = u_i$, and then

$$\omega_{n_i}^{l_i} = \omega_{n_i}^{k_i} \tag{2.41}$$

and so $l_i = k_i$ for all i, and $l = k$.

For the third part, we present two proofs. The first is neater, but the second has the advantage of showing where the ω come from.

For the first proof, note that there are exactly $|G|$ different ways to choose k in the construction above. All of those ways lead to different simple $\mathbb{C}G$-modules, and since by theorem 5 there are only (at most) $|G|$ simple $\mathbb{C}G$-modules then any simple $\mathbb{C}G$-module must be isomorphic to one of the above.

Alternatively, let M be a simple $\mathbb{C}G$-module. From theorem 7, M must then be one-dimensional as a $\mathbb{C}$-vector space. Consider the action of $v_{u_i} \in \mathbb{C}G$ on an element $a \in M$. There must be some $\lambda_i \in \mathbb{C}$ so that, for all a,

$$v_{u_i} a = \lambda_i a$$

and then

$$v_{2u_i} a = (v_{u_i} \star v_{u_i}) a = v_{u_i}(v_{u_i} a) = v_{u_i}(\lambda_i a) = \lambda(v_{u_i} a) = \lambda_i^2 a$$

and so on by induction up to

$$v_{n_i u_i} a = \lambda_i^{n_i} a$$

but $n_i u_i = 0$ as $u_i \in \mathbb{Z}_{n_i}$, and v_0 is the identity in $\mathbb{C}G$, so

$$\lambda_i^{n_i} = 1$$

that is, there is some $k \in G$ so that for all i

$$\lambda_i = \omega_{n_i}^{k_i}$$

and as any $g \in G$ can be expressed in the form

$$g = \sum_{i=0}^{m-1} g_i u_i$$

then g must have an action on the simple $\mathbb{C}G$-module of the form ω_G^{kg} and thus the simple $\mathbb{C}G$-module is isomorphic to $(\mathbb{C}, +, \cdot_k)$. $\square$

This allows us to write the multidimensional DFT explicitly.

2.5 The Multidimensional DFT

We will add to our notation by writing elements of $|G|\mathbb{C}$ also as vectors indexed by elements of G with the indexes ordered so that x_g corresponds to the simple $\mathbb{C}G$-module $(\mathbb{C}, +, \cdot_g)$ defined in theorem 8.

Then the multidimensional DFT F_G, an invertible $\mathbb{C}$-linear transformation $F_G : \mathbb{C}G \to |G|\mathbb{C}$, can be written explicitly, for $g, h \in G$, as

$$(F_G(v_h))_g = \omega_G^{gh} \tag{2.42}$$

When reduced to the one-dimensional case, this is exactly the definition given in the previous chapter.

2.6 The Multidimensional FFT

Now, the benefit of all this is that the one-dimensional DFT algorithms carry over, although with more complex behaviour.

Suppose $r, s, t, u \in G$ have the property that any elements $j, k \in G$ can be written uniquely in the form $j = ar + bs$ and $k = ct + du$, for $a \in A$, $b \in B$, $c \in C$ and $d \in D$ where $A, B, C, D \subseteq G$. Then we can write (we will omit the G in ω_G from now on)

$$(Fx)_j = \sum_{k \in G} \omega^{jk}(x)_k \tag{2.43}$$

as

$$(Fx)_{ar+bs} = \sum_{c \in C} \sum_{d \in D} \omega^{(ar+bs)(ct+du)}(x)_{ct+du} \tag{2.44}$$

The most common form of the multidimensional FFT is obtained by choosing r, s, t, u so that $r = u$ and $s = t$ and the components of r and s are all either 0 or 1. Then $A = D = rG$, $B = C = sG$. Also, $rt = 0$ in G, just as in the prime factor algorithm. From here, the derivation proceeds as in the previous chapter:

$$(Fx)_{ar+bt} = \sum_{d \in D} \sum_{c \in C} \omega^{(ar+bt)(ct+dr)} \tag{2.45}$$

$$(Fx)_{ar+bt} = \sum_{d \in D} \omega^{adrr} \sum_{c \in C} \omega^{(ac+bd)rt} \omega^{bctt} x_{ct+dr} \tag{2.46}$$

$$(Fx)_{ar+bt} = \sum_{d \in D} \omega^{ad} \sum_{c \in C} \omega^{bc} x_{ct+dr} \tag{2.47}$$

In short, to compute a multidimensional FFT, it is sufficient to compute one-dimensional FFTs over each of the dimensions.

Of course, this description includes the one-dimensional prime factor algorithm, since if n_0 and n_1 are coprime then $\mathbb{Z}_{n_0} \times \mathbb{Z}_{n_1}$ is isomorphic to $\mathbb{Z}_{n_0 n_1}$.

The algorithm here gives rise to a particular way of expressing the multidimensional FFT matrix as a tensor product of one-dimensional FFT matrices. The *tensor product* or *Kronecker product* of two matrices, written $M \otimes N$, is the matrix made up of blocks of the form $M_{jk}N$. Then it is possible to write

$$F_G = F_{\mathbb{Z}_{n_0}} \otimes F_{\mathbb{Z}_{n_1}} \otimes \ldots \otimes F_{\mathbb{Z}_{n_{m-1}}} \tag{2.48}$$

by using equation 2.47 repeatedly.

This is actually a special case of a much more general algorithm. In the general case, if a group G (not necessarily abelian) is the product of two groups so that $G = H \times K$, then the irreducible representations (the $\mathcal{M}_{f_i}$ of theorem 5) of G are the tensor products of those of H and K. Then the DFT matrix of G is the tensor product of that of H and K [5].

2.7 The Multidimensional Cooley-Tukey FFT

If the above is the multidimensional version of the prime factor algorithm, what about the Cooley-Tukey FFT?

Suppose, in addition to the property of r, s, t, u given above, we have $rt = 0$. Then we can write, exactly as in the one-dimensional Cooley-Tukey algorithm,

$$(Fx)_{ar+bs} = \sum_{d \in D} \omega^{adru} \omega^{bdsu} \sum_{c \in C} \omega^{acrt} \omega^{bcst} x_{ct+du} \tag{2.49}$$

$$(Fx)_{ar+bs} = \sum_{d \in D} \omega^{adru} \omega^{bdsu} \sum_{c \in C} \omega^{bcst} x_{ct+du}. \tag{2.50}$$

This leads to many multidimensional FFTs. A simple variant has $s = u = 1_G$, no component of r or t zero, and

$$A = D = \{a \in G : a_i < t_i \ \forall i\} \tag{2.51}$$

$$B = C = \{b \in G : b_i < r_i \ \forall i\} \tag{2.52}$$

which corresponds to performing the usual Cooley-Tukey FFT simultaneously on each dimension.

The benefit of this algorithm is that, as mentioned in section 1.9, it is better to use the Cooley-Tukey algorithm to reduce the FFT sizes before, rather than after, using the prime factor algorithm. This is true, for the same reasons, in the multidimensional case.

This algorithm, like many other FFT algorithms, has been discovered repeatedly. Some papers describing versions of this technique are [74, 86, 96]

[64] gives a unified treatement for the multidimensional case similar to the one here, although without the algebra. [6], by comparison, derives these results using a more algebraic approach than the above; the author recommends this paper for a more comprehensive treatment of the subject than the above two sections can give. [90] is an earlier paper than [6], for the one-dimensional case.

2.8 The Reduced Transform Algorithms

The Reduced Transform algorithms, variants of which are called *weighted redundancy transform* or *Gertner transform* [35], reduce a multidimensional FFT to a one-dimensional (or fewer-dimensional) FFT along subsets of G called 'lines'.

A *line* of G is a maximal cyclic subgroup of G. Every element of G must, of course, be contained in at least one line (the subgroup which it generates, if no other). Let $\Psi(G)$ be the number of lines in G; call the lines L_i for $0 \leq i < \Psi(G)$; let $l_i \in L_i$ be a generator of L_i, and let $W(g)$, $g \in G$, be the number of lines in which g is contained. Then the DFT can be calculated as, for integers h, i, j, k:

$$(Fx)_{hl_i} = \sum_{k=0}^{\Psi(G)} \sum_{j=0}^{|L_k|-1} \omega^{(hl_i)(jl_k)} \frac{x_{jl_k}}{W(jl_k)} \tag{2.53}$$

$$= \sum_{k=0}^{\Psi(G)} \sum_{j=0}^{|L_k|-1} \omega^{hj(l_i l_k)} \frac{x_{jl_k}}{W(jl_k)} \tag{2.54}$$

To further simplify this equation, let c be the least common multiple of $n_1, n_2, \ldots, n_m$. Let, for $g \in G$,

$$C(g) = \sum_i \frac{c}{n_i} g_i \tag{2.55}$$

so that, according to our notation,

$$\omega^g = \omega_c^{C(g)} \tag{2.56}$$

then

$$(Fx)_{hl_i} = \sum_{k=0}^{\Psi(G)} \sum_{j=0}^{|L_k|-1} \omega_c^{(hC(l_i l_k))j} \frac{x_{jl_k}}{W(jl_k)} \tag{2.57}$$

Now, this can be calculated by first computing $\Psi(G)$ one-dimensional FFTs of size c:

$$y(i,k) = \sum_{j=0}^{|L_k|-1} \omega_c^{ij} \frac{x_{jl_k}}{W(jl_k)} \tag{2.58}$$

then computing a sum of elements of y:

$$(Fx)_{hl_i} = \sum_{k=0}^{\Psi(G)} y(i, kC(l_i l_k)) \tag{2.59}$$

This algorithm was presented in [95, 93] in the version described above. It is possible to extend it from lines to planes or hyperplanes [91]; and [35] describes some special (but common) cases of the transform that eliminate the need for the weights $W(jl_k)$ and, by keeping all the lines the same size, can perform the summation in equation 2.59 on the input data.

The efficiency of the algorithm depends on the size of c and $\Psi(G)$. Generally, c has a much greater impact than Ψ; for the two-dimensional case where both dimensions are equal to some power p^t of a prime, $\Psi(\mathbb{Z}_{p^t}^2) = (p+1)p^{t-1}$ and the computation needs a little more than half the one-dimensional FFTs of the prime-factor-based algorithm presented above—better for larger p.

Unfortunately, the algorithm also has a more complicated data indexing pattern than the algorithms presented above, which can absorb any reduction in computation and even lead to a slower implementation overall.

2.9 The Finite Field FFT

Rader's algorithm for one-dimensional FFTs also extends to the multidimensional case. Consider, for prime p, $G = \mathbb{Z}_p \times \mathbb{Z}_p \times \ldots \times \mathbb{Z}_p$, where there are m copies of $\mathbb{Z}_p$. We can give G a field structure (the field is called $GF(p^m)$; the fields of the same size are isomorphic) by choosing a polynomial with coefficients in $\mathbb{Z}_p$, of order m, and which is irreducible, that is which cannot be written as a product of two nonconstantant polynomials with coefficients in $\mathbb{Z}_p$. Call the chosen polynomial q (it is not unique). Then G is a field where the multiplication is calculated by identifying elements $h \in G$ with polynomials $h'(x) = \sum_{i=0}^{m-1} h_i x^i$ and multiplying these polynomials modulo q. The addition in G is equivalent to adding the polynomials.

A well-known property [42] of finite fields is that the nonzero elements form a cyclic group under multiplication. Let $g \in G$ be a generator of this group. g is

usually not unique either, even for a fixed q; in fact the generators are sufficiently dense that trial and error is an effective way of finding one.

Now, this means that for each $x \in G$ so that $x \neq 0_G$, there exists some *integer* $0 \leq k < |G| - 1$, $x = g^k$. Since G is a field under the multiplication we are using, x^{-1} exists and $x^{-1} = g^{-k}$.

We can now apply this to the multidimensional DFT. The 0_G element must be treated specially:

$$(Fx)_0 = \sum_{i \in G} x_i \tag{2.60}$$

then for $1 \leq k \leq p^m - 1$,

$$(Fx)_{g^k} - x_0 = \sum_{j=0}^{p^m-1} \omega^{g^j g^{-k}} x_{g^{-k}} \tag{2.61}$$

$$= \sum_{j=0}^{p^m-1} \omega^{g^{j-k}} x_{g^{-k}} \tag{2.62}$$

This is (almost) identical to equation 1.41 in section 1.5. In particular, it is a convolution of the last $n - 1$ elements of x permuted by g^{-1} and the vector w of length $n - 1$ defined by

$$(w)_i = \omega^{g^{i+1}} \tag{2.63}$$

So this can also be computed as a one-dimensional convolution, and the remarks in the one-dimensional description of Rader's algorithm apply.

This algorithm was presented in [7].

There are a number of techniques that can be used to efficiently multiply elements in $GF(p^m)$; in fact, since polynomial multiplication is itself convolution, it is even possible to use a discrete Fourier transform to perform it, although this is likely to reduce efficiency because of increased overhead unless m is very large.

2.10 Bluestein's Algorithm in Multiple Dimensions

The equality used in Bluestein's algorithm,

$$jk = \frac{j^2 + k^2 - (j-k)^2}{2} \tag{2.64}$$

is almost, but not quite, valid in our group G; all is fine until we need to divide by 2, which may or may not be possible. Instead, we will note that

$$\prod_i \omega_{n_i}^{j_i k_i} = \prod_i \omega_{2n_i}^{j_i^2 + k_i^2 - (j_i - k_i)^2} \tag{2.65}$$

which is what is required, and then following section 1.6 we obtain

$$(Fx)_j = \left(\prod_i \omega_{2n_i}^{j_i^2}\right) \sum_{k \in G} \left(\prod_i \omega_{2n_i}^{(k-j)^2}\right) \left(\prod_i \omega_{2n_i}^{k^2}\right) x_k \tag{2.66}$$

which can be computed using a multidimensional convolution where the size of each dimension is doubled, as before.

The multidimensional algorithm is more efficient than applying Bluestein's algorithm to each dimension separately because it combines the componentwise multiplications for each dimension.

As before, the finite field FFT is faster than Bluestein's algorithm when it applies. In fact, since the finite field FFT only introduces one factor of 2, rather than 2^m, it can be much faster if a high-dimensional FFT is being performed.

2.11 Data Movement in the Multidimensional FFT

We can represent the most simple way of performing a multidimensional FFT by figure 2.2 (for the three-dimensional case); one-dimensional FFTs are performed over each dimension. Unfortunately, the sizes of the dimensions are often powers of two, which on a vector machine causes strided access and bank conflict when computing the FFT over the least significant index; and on a scalar machine causes suboptimal cache and virtual memory use when computing the FFT over the most significant index.

Thus, it is often done to insert two transpositions (only two, even for an arbitrarily large number of dimensions) to cause the one-dimensional FFTs to occur in the most convenient part of the index digits (figure 2.3). However, often the indices transposed will not be equal (just as in the Stockham algorithm in the one-dimensional case), leading to reduced efficiency and making the transposition difficult to perform in place without substantial effort and a significant performance penalty.

2.12 Square Transpose Multidimensional FFT

To reduce the performance cost of the transpositions, we propose a new algorithm which only uses square transpositions, based on the multidimensional Cooley-Tukey FFT, equation 2.50. The particular variation will be equivalent to applying equation 2.50 twice.

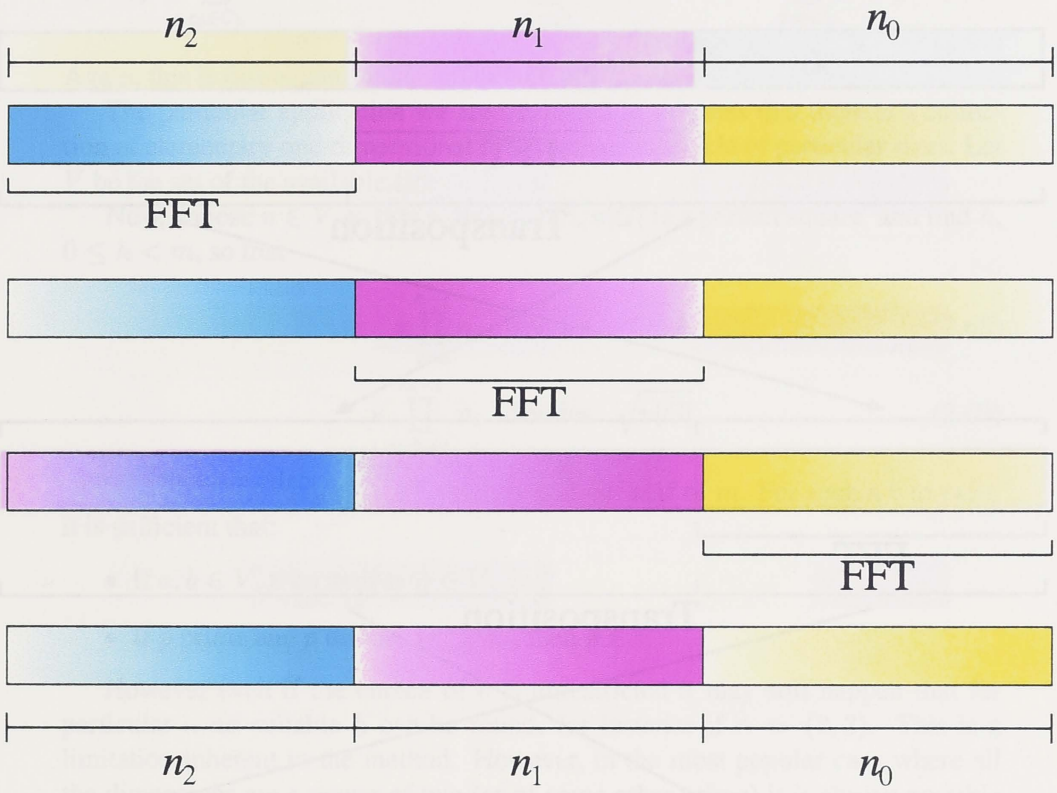


Figure 2.2: A simple way of computing a three-dimensional FFT.

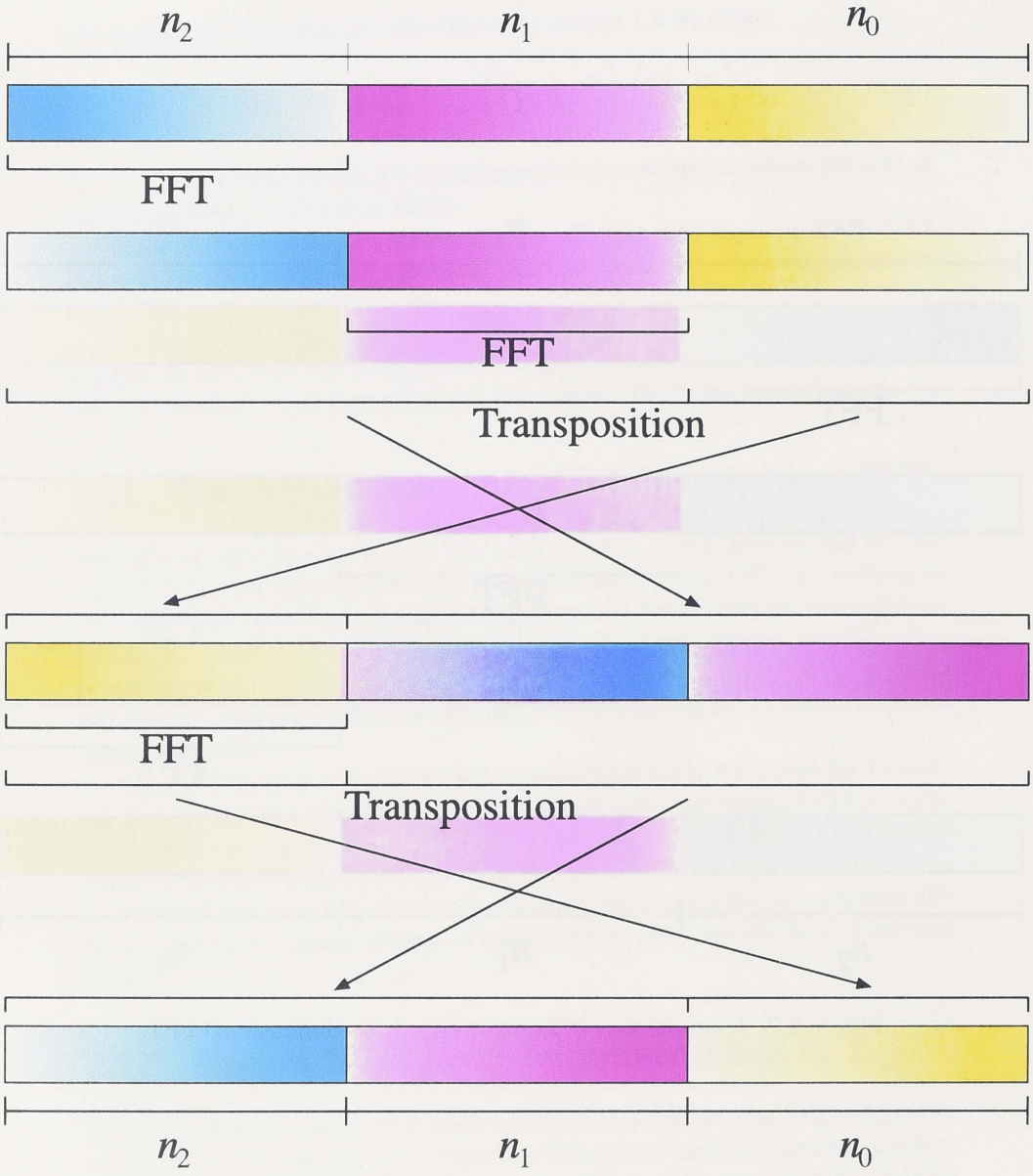


Figure 2.3: Computing a three-dimensional FFT with a transposition.

Figure 2.4 shows the following in the three-dimensional case.

Suppose there exist constants $r_0, r_1, r_2, t_0, t_1, t_2 \in G$ so that $r_0 t_0 = r_0 t_1 = r_1 t_0 = 0_G$ and some sets $A_i \subseteq G$ and $C_i \subseteq G$ so that any $j, k \in G$ can be written uniquely in the form $j = a_0 r_0 + a_1 r_1 + a_2 r_2$, $k = c_0 t_0 + c_1 t_1 + c_2 t_2$ for $a_i \in A_i$ and $c_i \in C_i$ for all i. Then by expanding and cancelling, we obtain:

$$(Fx)_j = \sum_{c_2 \in C_2} \omega^{a_0 r_0 c_2 t_2} \omega^{(a_1 r_1 + a_2 r_2) c_2 t_2} \sum_{c_1 \in C_1} \omega^{a_1 r_1 c_1 t_1} \omega^{a_2 r_2 c_1 t_1} \sum_{c_0 \in C_0} \omega^{a_2 r_2 c_0 t_0} (x)_k \tag{2.67}$$

Again, this is just equation 2.50, applied twice.

The particular application we are interested in assumes that there is a collection of elementary one-dimensional FFT routines available of particular sizes. Let V be the set of the available sizes.

Now, choose $v \in V$ so that v divides $|G|$, $v|G|$ is a perfect square, and find h, $0 \leq h < m$, so that

$$v \prod_{i=0}^{h-1} n_i \quad \text{divides} \quad \sqrt{v|G|} \tag{2.68}$$

$$v \prod_{i=h+1}^{m-1} n_i \quad \text{divides} \quad \sqrt{v|G|} \tag{2.69}$$

In the case where all the n_i are equal, h is about half of m. For such a v to exist, it is sufficient that:

- If $a, b \in V$, then $\gcd(a, b) \in V$; and
- If p prime and p divides $\prod_{i=0}^{m-1} n_i$, then $p \in V$.

However even if the choice of v is unrestricted it may still happen that for particular n no suitable h can be found, for instance if $n = (2, 3)$. This is a limitation inherent in the method. However, in the most popular case where all the dimensions are a power of two (or of some other prime) it *is* always possible to find a suitable tree.

Next, define $b_0 \in G$ to be the vector that is 1 on $0 \ldots h-1$, and 0 elsewhere; b_1 to be the vector that is 1 on h and 0 elsewhere, and b_2 to be the vector that is 1 on $h+1 \ldots m-1$ and 0 elsewhere. These form a basis for G (as a G-module) and the product of any pair is zero.

Also, let

$$l := \frac{\sqrt{v|G|}}{v \prod_{i=0}^{h-1} n_i}.$$

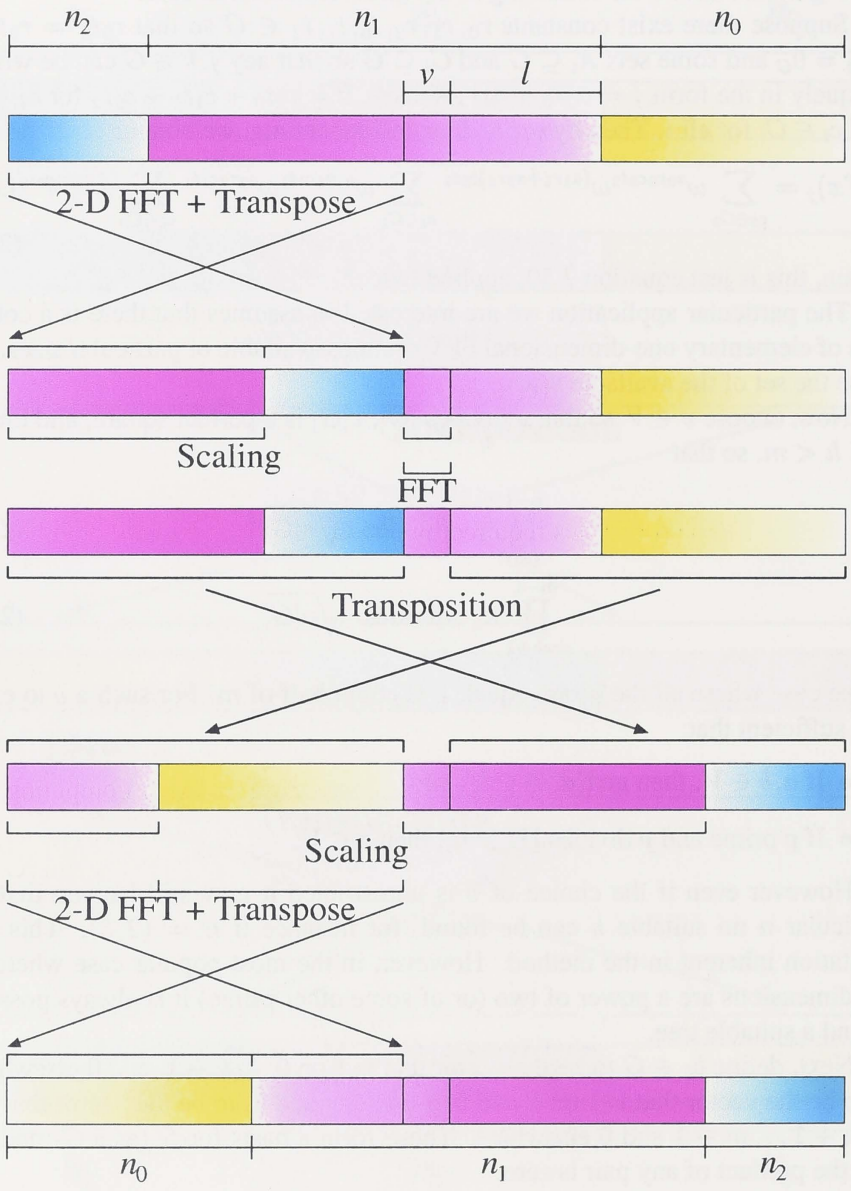


Figure 2.4: The square transpose method for three-dimensional FFTs.

Then, define

$$r_0 := b_0 + \frac{n_h}{l} b_1 \tag{2.70}$$

$$r_1 := \frac{n_h}{vl} b_1 \tag{2.71}$$

$$r_2 := b_1 + b_2 \tag{2.72}$$

$$t_0 := vlb_1 + b_2 \tag{2.73}$$

$$t_1 := lb_1 \tag{2.74}$$

$$t_2 := b_0 + b_1 \tag{2.75}$$

$$A_0 := \{c_0 \in (b_2 + b_1)G : (c_0)_h < \frac{n_h}{vl}\} \tag{2.76}$$

$$A_1 := \{c_1 \in b_1 G : (c_1)_h < v\} \tag{2.77}$$

$$A_2 := \{c_2 \in (b_1 + b_0)G : (c_2)_h < l\} \tag{2.78}$$

$$C_i := A_{2-i} \tag{2.79}$$

These have the properties required by equation 2.67, and in addition because of the way we have defined l,

$$|A_0| = |A_2| = |C_0| = |C_2| = \sqrt{\frac{|G|}{v}} \tag{2.80}$$

and $|A_1| = |C_1| = v$.

This allows us to perform the FFT by using the framework shown in figure 2.4, by performing a single square transposition and recursively computing two FFTs of smaller dimension.

2.13 Alternate Square Transpose Multidimensional FFT

The algorithm described in the previous section corresponds to the algorithm described in section 2.6. It is also possible to combine some of the componentwise multiplications, as described in section 2.7.

An example of a schema which does this, while still only using square transpositions, is shown in figure 2.5. The particular variant shown in the figure is for a $64 \times 64 \times 64$ FFT, and assumes the existence of an order-8 elementary FFT.

The method can be varied somewhat, depending on the elementary FFT sizes available. Generalisation to an FFT of size $ab \times b^2 \times ba$ is trivial if elementary

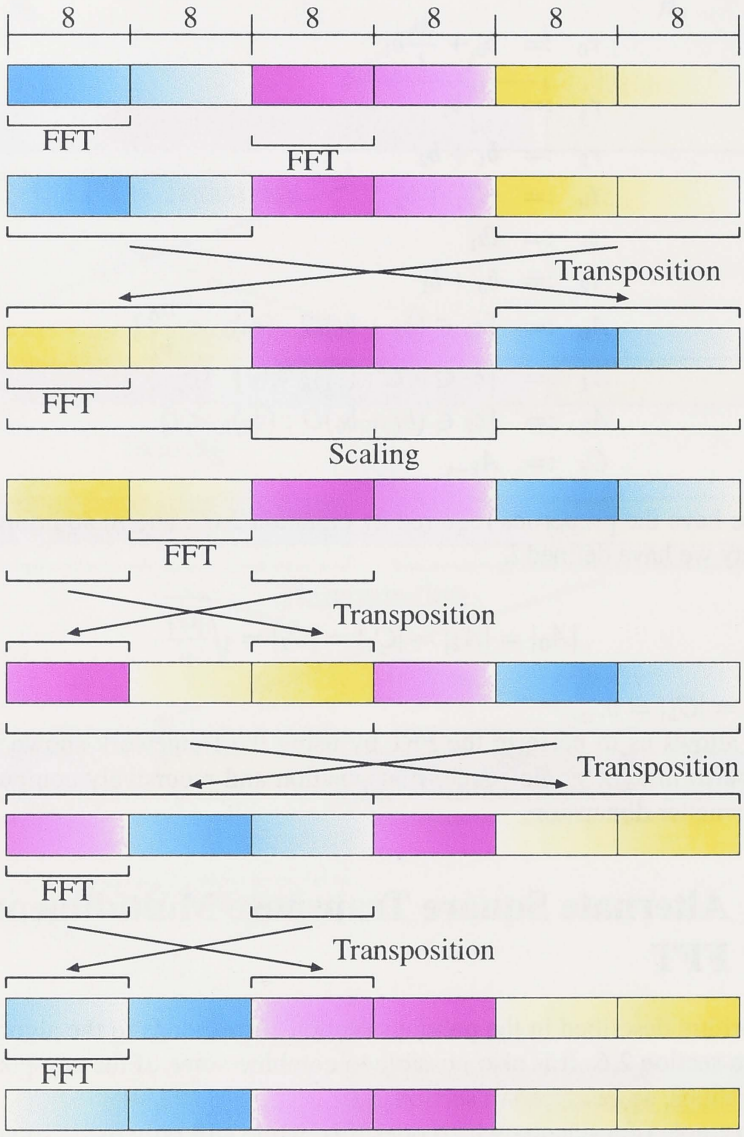


Figure 2.5: Square transpose method, with combined scalings, for $64 \times 64 \times 64$ FFT.

FFTs of size a and b are available. However, it is not claimed that this is a general algorithm.

The generalised form of this framework, for computing an FFT of size $ab \times b^2 \times ba$, is obtained by indexing X by a, b, b, b, a in that order and Y similarly, then computing

$$Z^{(1)}(\cdot, i, j, k, l, m) := F_a X(\cdot, i, j, k, l, m) \tag{2.81}$$
$$Z^{(2)}(h, i, \cdot, k, l, m) := F_b Z^{(1)}(h, i, \cdot, k, l, m) \tag{2.82}$$
$$Z^{(3)}(l, m, j, k, h, i) := Z^{(2)}(h, i, j, k, l, m) \tag{2.83}$$
$$Z^{(4)}(\cdot, m, j, k, h, i) := F_b Z^{(3)}(\cdot, m, j, k, h, i) \tag{2.84}$$
$$Z^{(5)}(l, m, j, k, h, i) := \omega^{lm+jk+hi} Z^{(4)}(l, m, j, k, h, i) \tag{2.85}$$
$$Z^{(6)}(l, \cdot, j, k, h, i) := F_a Z^{(5)}(l, m, j, k, h, i) \tag{2.86}$$
$$Z^{(7)}(j, m, l, k, h, i) := Z^{(6)}(l, m, j, k, h, i) \tag{2.87}$$
$$Z^{(8)}(k, h, i, j, m, l) := Z^{(7)}(j, m, l, k, h, i) \tag{2.88}$$
$$Z^{(9)}(\cdot, h, i, j, m, l) := F_b Z^{(8)}(\cdot, h, i, j, m, l) \tag{2.89}$$
$$Z^{(10)}(i, h, k, j, m, l) := Z^{(9)}(k, h, i, j, m, l) \tag{2.90}$$
$$Y(\cdot, h, i, k, m, l) := F_b Z^{(10)}(\cdot, h, k, j, m, l). \tag{2.91}$$

The framework also requires an extra transposition to that described in the previous section, and restricts the choice of elementary FFTs somewhat; so it will not necessarily be faster than the usual multidimensional square transpose algorithm. However, it also avoids reversing the order of the dimensions.

Chapter 3

An FFT Implementation for the Fujitsu VPP300

On its VPP series of vector-parallel supercomputers, Fujitsu provides a scientific software library (the SSLII/VP and SSLII/VPP libraries) parts of which, including the FFT routines, are developed by ANU. During the course of this thesis these routines were substantially enhanced (to the point that only a few subroutines still resemble the originals).

The library uses Cooley-Tukey and prime factor algorithms to compute FFTs whose length can be expressed as $2^a 3^b 5^c$ for any a, b, c. Additional prime factors can be easily added to the library. The library is designed to minimise the amount of workspace needed, to allow the computation of large FFTs or the computation of many FFTs at once, and is designed to be efficient in these cases.

The library also includes parallel FFT routines and includes routines which provide for real inputs or real outputs. There is also basic support for sine and cosine transforms. The non-complex FFTs are dealt with in the following chapter.

3.1 Structure of the VPP

The VPP300 consists of a variable number (1-16) of processing elements (PEs) linked together using a crossbar switch, so that each PE appears to be equidistant from the others. The ANU's VPP300 [27, 8] has 13 PEs, of which one is used for interactive work and so is not usually used for parallel jobs.

The PEs, diagrammed in figure 3.1, consist of a scalar unit, a vector unit, memory, and control and communication logic. Each PE supports up to 2GB of

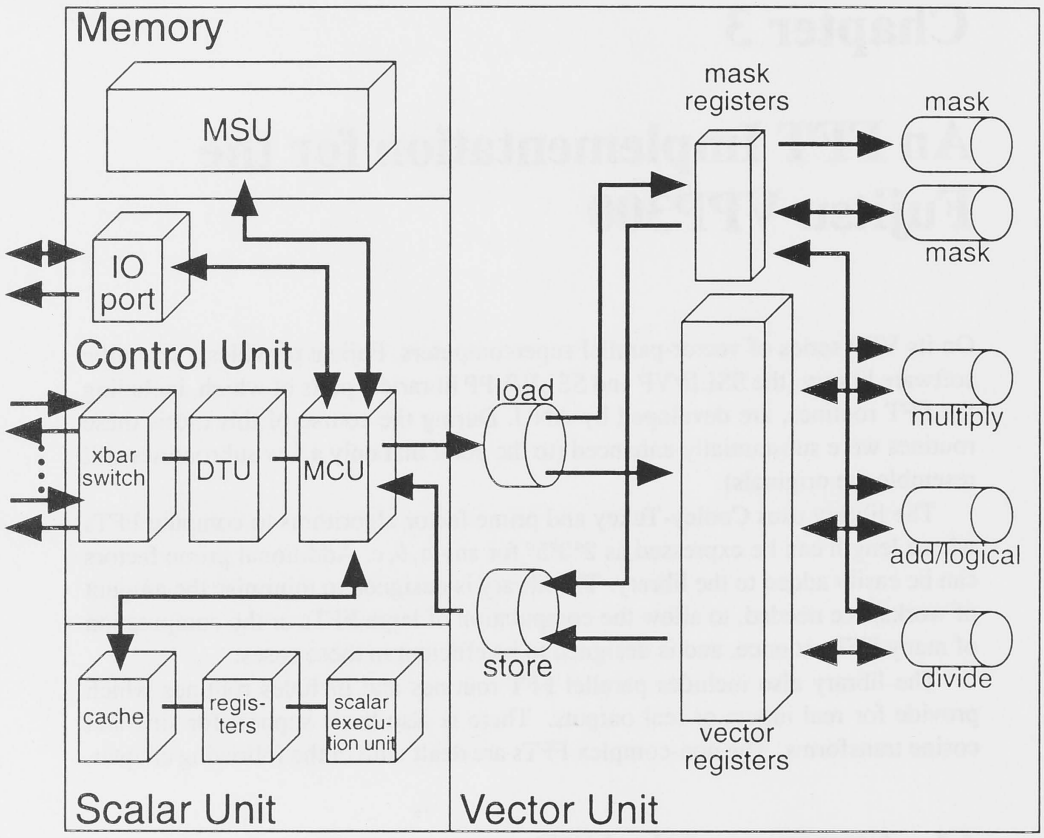


Figure 3.1: The structure of a single processing element on the VPP300 [44].

Table 3.1: Speed of memory access with increasing stride on the VPP300 [52].

Stride	2	3	4	5	8	16	32	64	128	255	256	512	1024
Time	3	3	10	3	11	18	30	60	60	3	110	240	260

'Time' is the average time for each memory access, expressed relative to the average time taken at stride 1, as measured on one particular loop.

memory (addresses are 32 bits wide), and can be linked to mass storage devices and networks. Each PE fits on a single circuit board.

The scalar unit is a reasonably conventional RISC processor, and can attain around 100MFlops. Unlike the vector unit, the scalar unit has a cache. We will not be greatly concerned with the inner workings of the scalar unit, instead treating it as something to be avoided for bulk computation.

The vector unit contains load, store, multiply, add, and divide pipelines, some mask logic which we will not be using, and the vector registers. The multiply and add pipelines can produce 8 results every 7ns clock cycle, for a total of 2.2GFlops. The divide pipeline produces 8 results only every 7 clock cycles, but fortunately we will not need to use it. Each pipeline has a start-up time, so performance degrades if calculations are performed with short vector lengths. The load, multiply, and add pipelines can be chained together; this reduces latency.

The vector registers hold 16384 floating-point values, and can be configured to range between 8 registers holding 2048 values to 256 registers holding 64 values each. The impact of this is that routines which use many temporary variables will have a shorter maximum vector length.

The VPP's memory is arranged in banks of 64 bytes each, to provide 8 double-precision values per clock cycle. The memory clock cycle is 10ns, so the memory bandwidth is not quite sufficient to feed the computational pipelines at top speed; this is part of the motivation for the banks. The ANU's VPP has 2GB of RAM on 5 PEs and 512MB on the other 8.

The load and store pipelines access data starting at some location with a *stride*. If the stride is 1, data is accessed sequentially at full speed. At other strides, data is accessed at approximately half-speed, and if the stride is divisible by a power of 2 then there is a penalty of approximately that power of 2, a very substantial performance degradation. For example, table 3.1 shows the time required for one particular loop. There is also a scatter/gather capability which the FFT code will not need.

3.2 Library Structure

Figure 3.2 shows the call structure of the FFT library, as implemented to use non-square transpositions when performing multidimensional FFTs. Figure 3.3 shows the call structure of the multidimensional FFT that uses square transpositions.

The routines in the library are modified before inclusion in the SSLII library, to avoid conflicts, but for development purposes follow a consistent pattern: internal routines are named starting with `dc`, for double-precision complex, followed by a short description of the routine; external routines are of the form `dvXYft` for "double-precision vector Fourier transform". `X` is either `m`, for multidimensional FFT or `s` for single-dimensional, and `Y` is the initial letter of complex, real, sine, or `o` for cosine transform, reflecting the symmetry in the input data.

The basic data structure the library operates on is the vector of complex numbers. To avoid the stride-2 access that would occur if Fortran's `complex` type was used, these vectors are represented using two vectors of double-precision values, one to hold the real part and one to hold the imaginary part. Although the library very often treats these vectors as multidimensional arrays, index computation is usually performed explicitly so as to avoid confusion about the current shape of the array.

When a complex vector is passed to a routine and it is to be treated as a multidimensional array, the dimensions of the array are passed in variables `n0`, `n1` and so on, where `n0` is the least-significant dimension. This means that the first choice for vectorisation is over `n0`, although for most routines a complex set of heuristics is used to determine which dimension to vectorise over.

The library does not include a version for single-precision values. Computing a 1024-element forward and inverse transform in succession in single-precision produces results accurate to as few as 7 bits, and it is expected that this will rarely be useful.

The library is written in Fortran 90. Almost all the code is compatible with an extended Fortran 77.

3.3 Library Interface

The user interface to the library has been carefully designed to be simple, easy-to-use, and efficient. The manual pages for the routines are attached as an appendix.

Some of the significant features are:

- Only two entry points, one of which is a simplified version of the other.

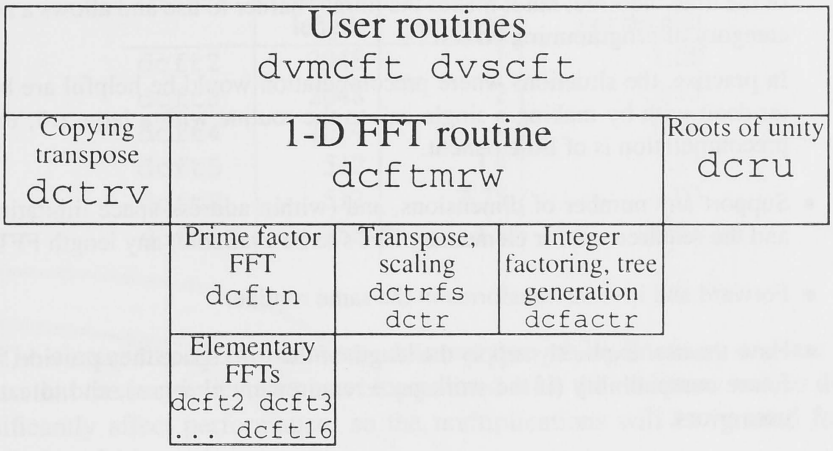


Figure 3.2: Structure of the FFT library.

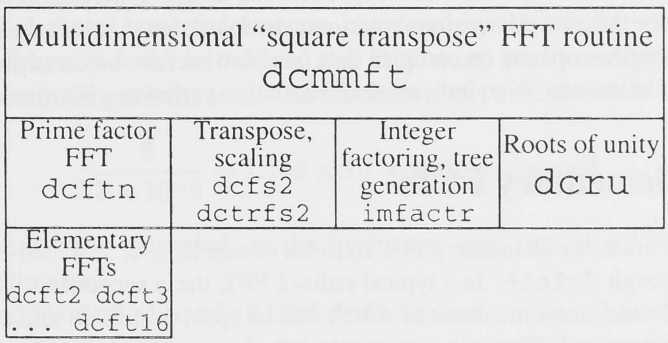


Figure 3.3: Structure of the 'square transpose' multidimensional FFT implementation.

- No need to pre-compute ω values or a description of the schema to use. Consideration was given to allowing this, as it would save up to about 20% of the time required; but it makes the library harder to use and allows a new category of programming errors.

  In practise, the situations where precomputation would be helpful are better dealt with by making a single call to the routine with a large `n0`; then precomputation is of little benefit.

- Support any number of dimensions, and (within address-space limitations and the requirement for elementary FFTs to be available) any length FFT.

- Forward and inverse transforms in the same routine.

- Have the user explicitly supply the length of the workspace they provide, for future compatibility (if the workspace requirement changes), and to catch user errors.

- Supply detailed error responses to the user via an integer output parameter `icon`.

- Regular naming convention; as provided to Fujitsu, the user-callable routines are named in a consistent way. The first letter of the routine name is 'd', for double-precision; the second letter is 'v' for the vector versions of the routines; the third letter is 's' for the single-dimensional routines or 'm' for the multidimensional routines; the fourth letter is 'c' to indicate that the routines operate on complex data (see also section 4.4); and the fifth and sixth letters are 'ft' to indicate that the routine performs a Fourier transform.

3.4 Elementary FFTs

The library includes elementary FFT routines of size 2, 3, 4, 5, 8, and 16, named `dcft2` through `dcft16`. In a typical radix-2 FFT, these account for about 40% of the total execution time, most of which will be spent in `dcft8` and `dcft16`.

The elementary FFT routines are combined using a prime factor algorithm into the routine `dcftn` which can perform FFTs of sizes of the form $2^a 3^b 5^c$ where $0 \leq a \leq 4$ and $b, c \in \{0, 1\}$. Additional sizes of elementary FFT can be easily added by changing only `dcftn`. `dcftn` performs no significant computation itself, it simply calls the elementary FFTs with the appropriate parameters.

Table 3.2: Performance of elementary FFT routines, $n_0 = 65536$.

Routine	Register length	Additions per word	10^9 operations per second
`dcft2`	2048	1	1.00
`dcft3`	2048	2	1.15
`dcft4`	1024	2	0.98
`dcft5`	512	3.4	1.11
`dcft8`	512	3.25	1.03
`dcft16`	256	4.5	0.91

The elementary FFTs use more additions than multiplications, and are written so that the multiplications can be pipelined with the additions and so do not significantly affect performance; so the multiplications will be ignored for the remainder of this section.

Table 3.2 gives the number of additions performed on each double-precision word in the input (so, for instance, `dcft2` actually performs four floating-point additions for each element in its vector). For these routines, the first addition operates on two values in memory. Thus, there are two additional operations to the loop: an initial load (the next load can be pipelined with the first addition), and a final store of the result of the last addition.

Figure 3.4 shows how performance, measured by the number of 'operations' (that is, additions plus the initial load and final store) performed per second, improves as vector length increases. The theoretical maximum here is

$$\frac{8}{7 \times 10^{-9}} = 1.143 \times 10^9 \text{ operations/second}$$

This can be slightly exceeded, as the final store may sometimes be overlapped with the initial load.

The routines `dcft4`, `dcft8`, and `dcft16` do not come close to this performance. Investigation of the produced assembly code indicate that the compiler is not scheduling the code in what would appear to be the optimal way.

For instance, figure 3.5 shows the scheduling for `dcft4`, as generated by the Fujitsu compiler (note that computing an FFT of order 4 requires no multiplications). Instructions that execute simultaneously are shown on the same line, and execution proceeds from top to bottom; arrows indicate a data dependency within

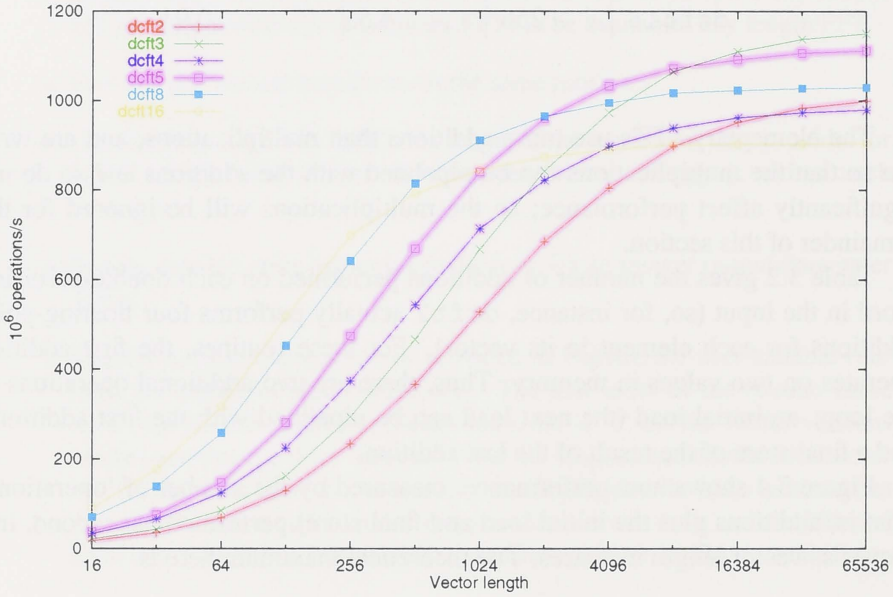


Figure 3.4: Performance of `dcftn` with varying vector lengths.

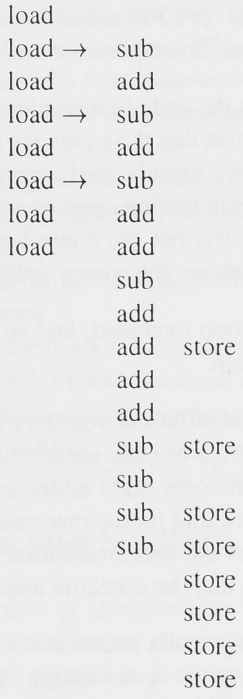


Figure 3.5: Scheduling graph for `dcft4`.

a line. There is a 'tail' of four store operations, where instead one would expect to see all but one of the store operations overlap the addition operations. Appendix C shows example source code for the inner loop of `dcft4` which, as written, is scheduled in what is believed to be an optimal way, but which the compiler schedules as in the figure.

Similarly, `dcft8` has a tail of 6 store operations, and `dcft16` has a tail of 15 store operations.

The low-level details of the VPP300 are undocumented, which makes it difficult to determine why this is so. Some possible reasons include:

- The compiler schedules the code in a single pass, starting with the first instruction, and cannot detect that the stores are all to different locations. This would mean that the stores must be performed in the order they are written, and so the addition and subtraction operations that feed the stores must also be performed in that order; but the compiler schedules the additions and subtractions first, and chooses the wrong order.

- The stores overlap the loop overhead, and so scheduling them in the suggested way would not help.

- The compiler has run out of scalar registers in which to compute the addresses of the loads and stores (this seems rather strange, given that there are 32 scalar registers and only eight addresses to keep track of, but it appears to actually happen), and then either cannot schedule scalar and vector operations together, or the synchronisation overhead between scalar and vector operations means that the schedule used is optimal.

The code presented in the appendix seems to need only eight vector registers, but the compiler uses 16, halving the vector length; `dcft5` is also using a shorter vector length than necessary. This is also an effect of the scheduling process, and is a difficult problem to solve in a modular compiler; one needs to perform scheduling, common subexpression elimination and register allocation together in one extremely complicated process to be able to avoid this, and in practise this is very hard. On the VPP the performance cost is significant, as indicated by the relative large input performance of `dcft16` compared with `dcft8` in figure 3.4, for instance.

This problem was also found by Frigo [32] on scalar machines. On a scalar machine, the effect is even more pronounced, because the number of registers available is fixed and so any rearrangement of the code to require more temporary

values than registers causes some registers to be temporarily stored ('spilled') in memory and re-loaded again later. Frigo solved this by causing the scheduling to happen after registers were allocated, but this does not seem to be possible with the Fujitsu compiler on the VPP.

On the VPP, `dcft16` requires the maximum of 64 vector registers, which implies that a `dcft32` routine (if one was written) would need to spill registers to memory, at a further cost in performance. A `dcft32` routine, if written, would still have a significant speed advantage over, say, performing `dcft8`, a transposition and scaling, and `dcft4`; but for large FFTs, the comparison is instead between five `dcft16` iterations (with four scale-and-transpose executions) instead of four `dcft32` iterations with three scale-and-transpose executions, which is a much reduced advantage.

The relatively poor performance of `dcft2` may be related to the high memory bandwidth it requires. Unlike all the other routines, `dcft2` needs to perform simultaneous loads and stores.

For shorter vectors, the performance is determined by loop overhead and the start-up time of vector operations.

Some earlier work on optimising these routines, on the VPP500 [94] (the VPP500 is a predecessor of the VPP300 that these results are based on) indicates an alternative technique which may allow more explicit control of the compiler's scheduling; it can be found in [46].

3.5 Transpositions and Scalings

The library requires a collection of transposition and componentwise multiplication routines. The various routines are shown in figures 3.6 and 3.7. Of the routines, `dctrfs2` is the most general; in fact, `dctrfs2` calls `dctrfs`, `dctr`, and `dcfs2` in the appropriate circumstances.

The routines take about 35% of the total execution time in a typical FFT, about half of this is accounted for by the single transposition and scaling that occurs at the top level of the recursion (that is, when $n_0 = 1$).

Because of this, the routines have been carefully implemented to attain optimum performance. Depending on the parameters, `dctrfs2` can use one of five methods, one of which is to perform the transposition and scaling separately—and there are seven scaling methods and three transposition methods, although some of those would never be used from `dctrfs2`.

In what follows, we will refer to the vector indices in figure 3.7, n_0 through

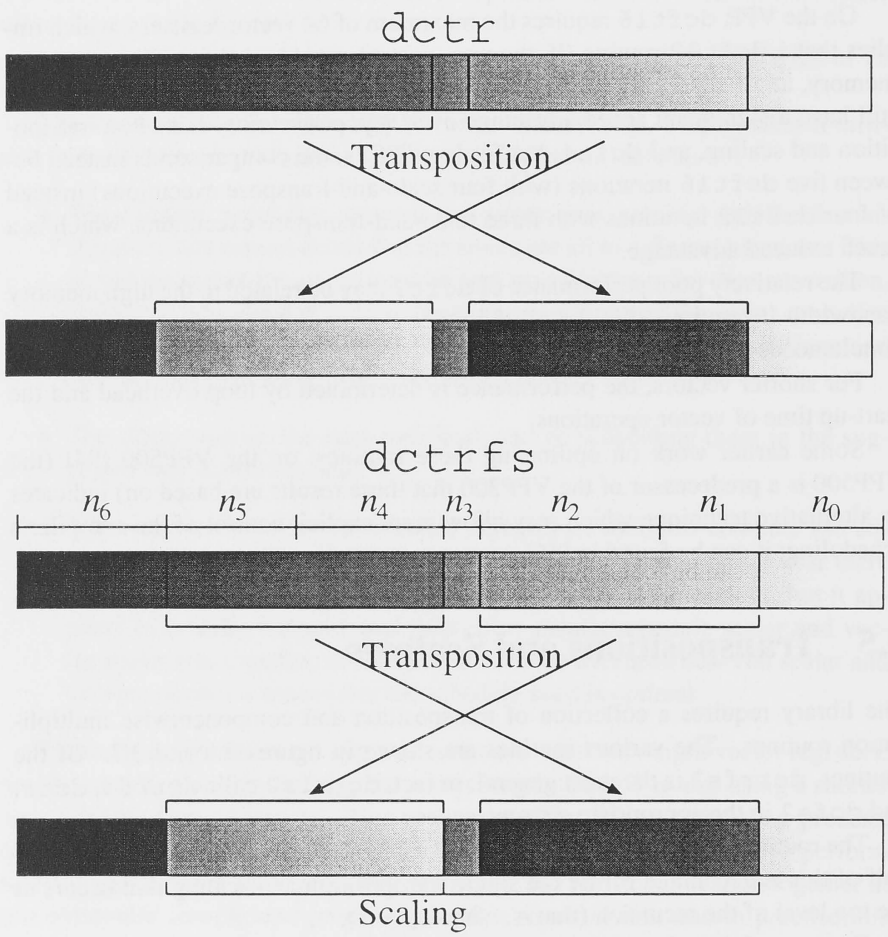


Figure 3.6: Operation performed by `dctr` and `dctrfs`.

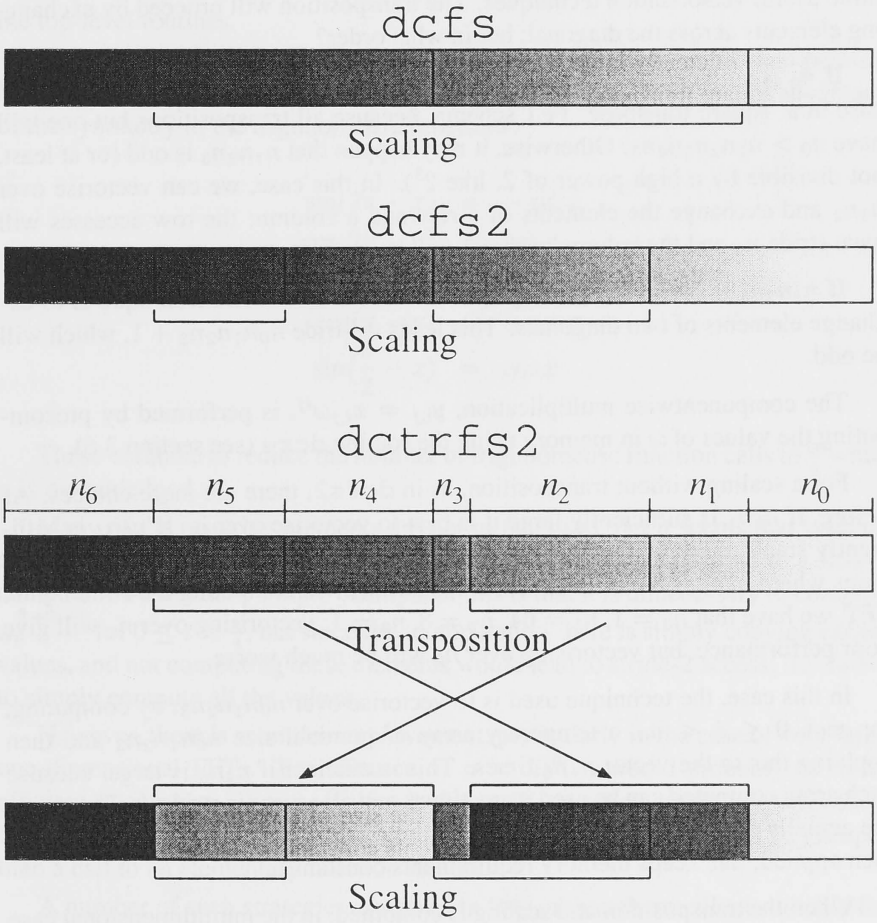


Figure 3.7: Operation performed by `dcfs`, `dcfs2`, and `dctrfs2`.

n_6. n_0 is the least significant index; the transposition is performed over n_1n_2, n_4n_5 and the componentwise multiplication ('scaling') over n_2n_3, n_5.

In the simplest case, that of a transposition (as in `dctr`), there are essentially three useful vectorisation techniques. The transposition will proceed by exchanging elements across the diagonal; but in what order?

If n_0 is sufficiently large it is best to vectorise over it; this will be the usual case in a 'square transpose' FFT schema, because all transpositions but one will have $n_0 > n_1n_2n_3n_4n_5$. Otherwise, it may happen that $n_1n_2n_3$ is odd (or at least, not divisible by a high power of 2, like 2^3). In this case, we can vectorise over n_1n_2 and exchange the elements of a row and a column; the row accesses will have stride n_0 and the column accesses will have stride $n_0n_1n_2n_3$.

If $n_1n_2n_3$ is divisible by a high power of 2, then a better technique is to exchange elements of two diagonals. This leads to stride $n_0n_1n_2n_3 + 1$, which will be odd.

The componentwise multiplication, $y_{i,j} = x_{i,j}\omega^{ij}$, is performed by precomputing the values of ω in memory using the routine `dcru` (see section 3.6).

For a scaling without transposition, as in `dcfs2`, there are more choices. As before, if n_0n_1 is sufficiently large it is best to vectorise over it. If n_0n_1 is sufficiently small, then it should be effective to vectorise over n_2n_3. This leaves the cases where n_0n_1 is intermediate—for instance, when computing a $64 \times 64 \times 64$ FFT, we have that $n_0 = 1, n_1 = 64, n_2 = 8, n_3 = 1$; vectorising over n_1 will give poor performance, but vectorising over n_2 will be much worse.

In this case, the technique used is to vectorise over $n_0n_1n_2n_3$, by computing, for each $0 \leq i < n_5$, a temporary array of nominal size $n_0n_1n_2n_3$ and then applying this to the vector n_4n_6 times. This is efficient if n_4n_6 is large, because each array computed can be used many times, and all access is stride-1. The arrays are actually computed in chunks of about the size of a vector register, which are then applied; this keeps memory requirements constant.

When the transposition and scaling is combined, in the multidimensional case there is only one reasonable vectorisation choice, to vectorise over n_0; otherwise, the transposition and scaling will be more efficient when split up. As vectorisation over n_0 will be suitable for all but one transposition/scaling, this has bounded performance impact. In the one-dimensional case (that is, when $n_2 = n_4 = 1$), the techniques used for vectorising transpositions will work just as well when a scaling is performed simultaneously.

3.6 Generating Roots of Unity

`dcru` computes ω_n^i for $i = 0, 1, \ldots n-1$ into an array in memory. This array is the only workspace required by the FFT routines, and is filled once at the start of the top-level routines.

At present, `dcru` is not very sophisticated, although it is robust and precise. It simply uses the `sin` and `cos` intrinsics (as in equation 1.5), taking advantage of the symmetry in the trigonometric functions:

$$\begin{aligned}
\sin(2\pi - x) &= -\sin x \\
\cos(2\pi - x) &= \cos x \\
\sin(\pi - x) &= \sin x \\
\cos(\pi - x) &= -\cos x \\
\sin(\frac{\pi}{2} - x) &= \cos x
\end{aligned}$$

These combine to reduce the number of trigonometric function calls to $\frac{n}{4}$ when n is divisible by 4.

Many of the resulting elements are not in fact used, but it seems that keeping careful track of which would be used would be complicated and lead to a slower computation overall. A simple example of this is that it is unnecessary to compute $\omega_n^{\frac{n}{2}+2i+1}$ for $0 \leq i < \frac{n}{4}$, but since the 'computation' here is simply copying earlier values, and not computing these elements would lead to stride-2 access, it is faster to simply compute all the values.

However, there is room for improvement, particularly in the case of very large one-dimensional FFTs. For instance, if $n = 2^{20}$, it would probably be more efficient to compute ω_n^j and $\omega_n^{2^9 j}$ for $0 \leq j < 2^9$, and then compute more values of ω by writing $\omega^{2^9 j+k} = \omega^{2^9 j}\omega^k$, as a complex multiplication would be cheaper than a call to an elementary function routine.

A number of such strategies are given in [61], although not all are suitable for vector implementation. It should be noted that because `sin` and `cos` vectorise on the VPP300, it is certainly not useful to use these techniques when only a small number (say, less than 2^{15}) coefficients are to be calculated.

One reason why these strategies are not used is that in the case of an FFT of size 2^{20}, only about 10% of the time is spent in `dcru`. It therefore seems pointless to optimise this routine further.

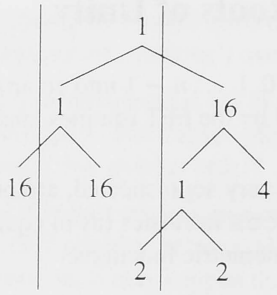


Figure 3.8: Tree for $128 \times 32 \times 16$ FFT.

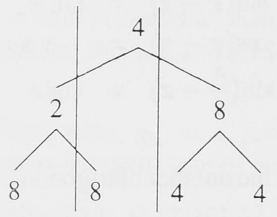


Figure 3.9: Alternative tree for $128 \times 32 \times 16$ FFT.

3.7 Trees

In the square transpose algorithms, it is necessary to choose the size of the elementary FFT under various constraints (this size was called v in section 2.12). When elementary FFTs are available for second or higher powers of a prime, there may be more than one choice for v available. The choices of c in each recursive call can be formed into a tree which fully describes the invocations of the algorithm. Figure 3.8 shows an example of such a tree; the vertical lines separate the nodes of the tree corresponding to different dimensions.

The computation represented by this tree proceeds from left to right in order. First, there is an elementary FFT of order 16, then a (trivial) 16×1 scaling, then a (trivial) elementary FFT of order 1, then a $16 \times 1 \times 16$ transposition, then a (again, trivial) 1×16 scaling, then another elementary FFT of order 16, then a (trivial) 16×1 scaling, then a $256 \times 1 \times 256$ transposition, then a 16×2 scaling, and so on.

It is not necessarily true that there is only one possible tree that can be used to calculate a particular DFT. For example, figure 3.9 shows another possible tree which could be used to perform the same FFT.

Naturally, we wish the fastest possible FFT. A count of the number of operations required for each step of the algorithm in our implementation is presented has been given in the previous sections; table 3.2 gives the operation count for the elementary FFTs, and there are two operations per word for each componentwise multiplication executed.

With this information, we can estimate the cost of the tree in figure 3.8 to be $27.5|G|$ operations (here, $|G| = 2^{17}$) while the cost of the tree in figure 3.9 is $30.75|G|$ operations. Clearly, we want to use the first tree!

A straightforward (although somewhat complicated) algorithm for finding the most efficient tree is presented in figure 3.10. The algorithm computes the cost for all trees that would lead to an FFT.

This algorithm initially calculates the lowest cost; the tree can be determined by the values of c' produced by each call of the algorithm. If g is ∞ at the end of the algorithm, no suitable tree exists.

In the algorithm, all the divisions and square roots produce exact integer results. c divides u, and u divides both $\frac{an_k}{e}$ and $\frac{n_k e}{a}$ from line 6, so the divisions on lines 10 and 12 are exact. In fact, c divides n_k which ensures that the elementary DFT need only be one-dimensional. The divisions on line 6 are guarded by the tests on line 3.

To show that the square roots are exact, we need this small result:

Theorem 9 *If* $\gcd(x, y)$ *is a perfect square, and* xy *is a perfect square, then* x *and* y *are perfect squares.*

To show this, write $x = \gcd(x, y)x'$, and similarly $y = \gcd(x, y)y'$. Then $xy = (\gcd(x, y))^2 x'y'$ so $x'y'$ must be a perfect square. But x' and y' are coprime, so x' and y' must also be perfect squares; then x is the product of two perfect squares, so must be a perfect square; similarly for y. □

Now, notice that $\frac{u}{c} = \gcd\left(\frac{n_k e}{ac}, \frac{an_k}{ec}\right)$ because the divisions are exact. So, because $\frac{u}{c}$ is a perfect square, and $\frac{n_k e}{ac}\frac{an_k}{ec} = \left(\frac{n_k}{c}\right)^2$ is also a perfect square, then the theorem above implies that the square roots on lines 8 and 10 are exact.

Furthermore, note that $ab = de = \sqrt{\frac{an_k e}{c}}$. This guarantees that the transpositions will be square.

In practise, of course, it is somewhat expensive to repeatedly compute GCD operations and divisions. Since elements of C are often powers of a small set P of

Require: C the set of elementary FFTs we can compute;
Require: $n_1, n_2, \ldots, n_m$ the dimensions of the FFT to be evaluated; $n_i > 1$.
Ensure: The lowest cost of computing an FFT of dimensions $n_1, \ldots, n_m$ is g.

1: Find k so that $a < n_k e \leq n_k^2 a$, where $a = \prod_{i=1}^{k-1} n_i$ and $e = \prod_{i=k+1}^{m} n_i$.
2: $g \leftarrow +\infty$
3: **if** a does not divide $n_k e$, or e does not divide $a n_k$ **then**
4: {No suitable tree exists.}
5: **else**
6: Let $u = \gcd\left(\frac{n_k e}{a}, \frac{a n_k}{e}\right)$.
7: **for all** $c \in C$ so that c divides u and $\frac{u}{c}$ is a perfect square **do**
8: $b \leftarrow \sqrt{\frac{n_k e}{ac}}$
9: $h_l \leftarrow$ the lowest cost of computing an FFT of dimensions $n_1, n_2, \ldots, n_{k-1}, b$ (a recursive operation)
10: $d \leftarrow \sqrt{\frac{a n_k}{ce}}$
11: $h_r \leftarrow$ the lowest cost of computing an FFT of dimensions $d, n_{k+1}, n_{k+2}, \ldots, n_m$ (a recursive operation)
12: $h_t \leftarrow h_l + h_r +$ the cost of computing the scaling, transposition, and elementary FFT
13: **if** $h_t < g$ **then**
14: $g \leftarrow h_t$
15: $c' \leftarrow c$
16: **end if**
17: **end for**
18: **end if**

Figure 3.10: Multidimensional tree-generation algorithm

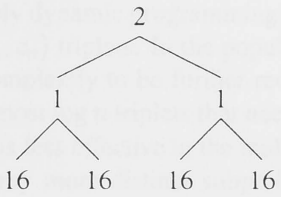


Figure 3.11: Optimal tree for FFT of size 2^{17}.

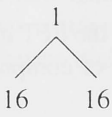


Figure 3.12: Optimal tree for FFT of size 256.

primes, it may be better to write the n_i, a through e, and u, as vectors (n_i) over P so that $n_i = \prod_{p \in P} p^{(n_i)_p}$. Then the loop on line 7, for instance, is over all vectors $(c) \in C$ so that $(c)_p \leq (u)_p$ and $((c)_p \bmod 2) = ((u)_p \bmod 2)$. This is what is done in our implementation.

It is particularly important to contain the cost of the tree computation on the VPP, because it is not vectorisable and so each operation needed here takes about as much time as ten arithmetic operations. The current version takes less than 1% of the total time required for a 2^{20} FFT.

The depth of the recursion (and the maximum height of the tree) is of order $O(\log \log N)$, where $N = \prod_{i=1}^{m} n_i$, because in each recursive call N is no more than the square root of the value in the caller, and an FFT of size 1 has cost 0. Since each invocation of the algorithm makes no more than $2|C|$ calls, the total algorithm takes time of $O((2|C|)^{\log \log N})$ or

$$O\left((\log N)^{\log(2|C|)}\right). \tag{3.1}$$

In the one-dimensional case, we can further simplify the computation by observing that any subtree of an optimal tree must also be an optimal tree (for a smaller FFT, of course). For example, the left and right subtrees of a tree of size 2^{17} (figure 3.11) must both be optimal (figure 3.12). This allows us to remove the constant 2 in equation 3.1, producing the algorithm shown in figure 3.13.

Require: C is the set of all elementary FFTs we can compute;
Require: n is the size of the FFT to be evaluated, $n > 1$.
Ensure: The lowest cost of computing an FFT of size n is g_n.

```
1:  a ← 1
2:  e ← 1
3:  for all c ∈ C so that c divides n and n/c is a perfect square do
4:      b ← sqrt(n/c)
5:      h_l ← the lowest cost of computing an FFT of size b.
6:      d ← b
7:      h_t ← 2h_l + the cost of computing the scaling, transposition, and elementary FFT
8:      if h_t < g then
9:          g_n ← h_t
10:         c_n ← c
11:     end if
12: end for
```

Figure 3.13: One-dimensional tree-generation algorithm

It is then possible to apply dynamic programming [25, chapter 16] by keeping a record of computed (n, g_n, c_n) triplets. In the popular case of $n = 2^r$, dynamic programming allows the complexity to be further reduced from $O((\log n)^{\log |C|})$ to $O(\log n)$, as there are at most $\log n$ triplets that need to be generated.

Dynamic programming is less effective in the multidimensional case, because in typical use there are many more distinct subproblems that must be solved. For the VPP implementation, dynamic programming is used only for the one-dimensional case.

The VPP implementation is the routine `imfactr`, written as a pair of recursive routines in Fortran 90. The new features in Fortran 90 made it possible to handle the complex data structures required with significantly more ease than would have been the case in older Fortran dialects.

3.8 Testing

An advantage of the highly modular structure of the library is that each part is relatively simple and can be tested (and, importantly, debugged) independently during construction. For regression testing, however, it is easier to test the top-level routines in such a way as to ensure all parts of the lower-level routines are exercised.

The test cases consist of two classes. Some are known answer tests, in which a transform of a vector is compared against an explicitly computed DFT (usually the vector is a basis vector, to simplify the DFT computation). The remainder compute the forward then inverse transform of a pseudo-random input vector, which should be (approximately) the identity function.

One important feature of the second class of tests is that the workspace is filled with random values between the computation of the forward and inverse transform. This ensures that all the required values are really being computed in the workspace, not obtained from the data left there by the forward transform.

3.9 Performance

The new multidimensional 'square transpose' technique does involve a trade-off, between the complexity of the topmost scaling (and the tree computation) against better vector lengths for the remainder of the algorithm. One important question is whether this trade-off was successful.

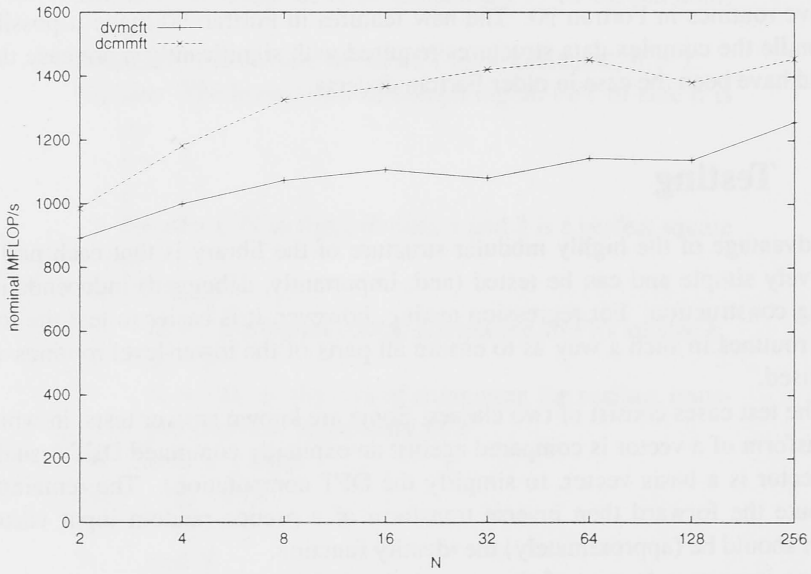


Figure 3.14: Performance comparison of multidimensional FFT algorithms for FFT of size $128 \times 128 \times N$.

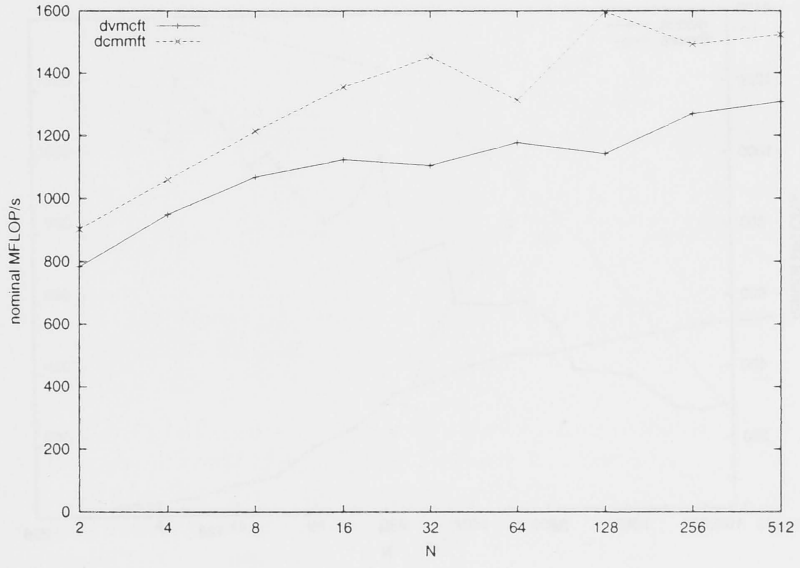


Figure 3.15: Performance comparison of multidimensional FFT algorithms for FFT of size $128 \times 64 \times N$.

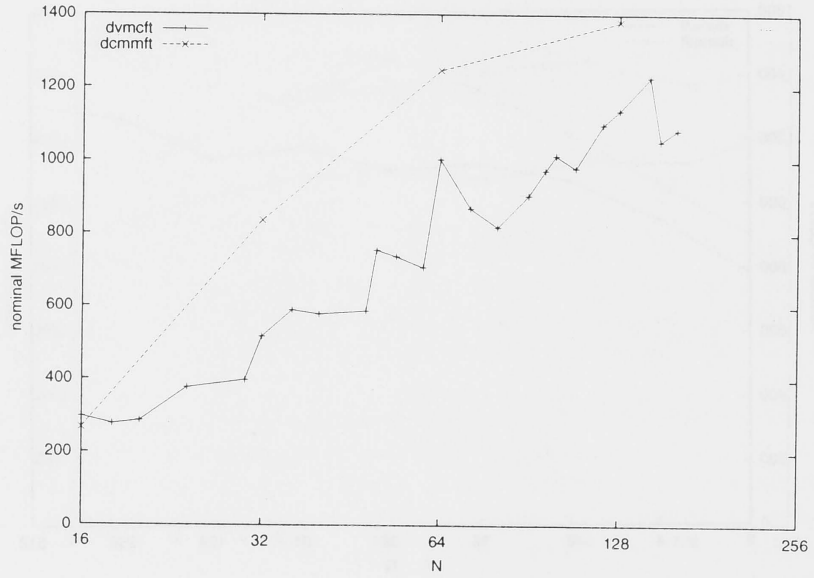


Figure 3.16: Performance comparison of multidimensional FFT algorithms for FFT of size $N \times N \times N$.

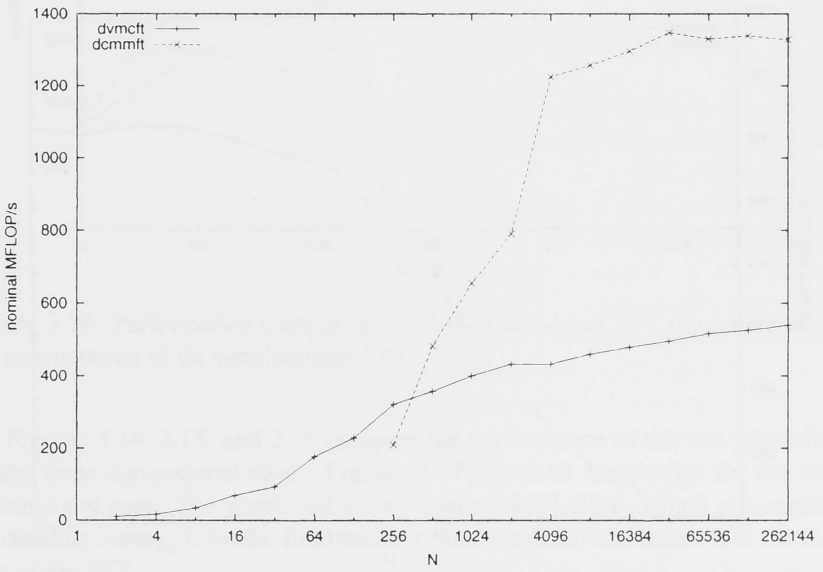


Figure 3.17: Performance comparison of multidimensional FFT algorithms for FFT of size $16 \times N$.

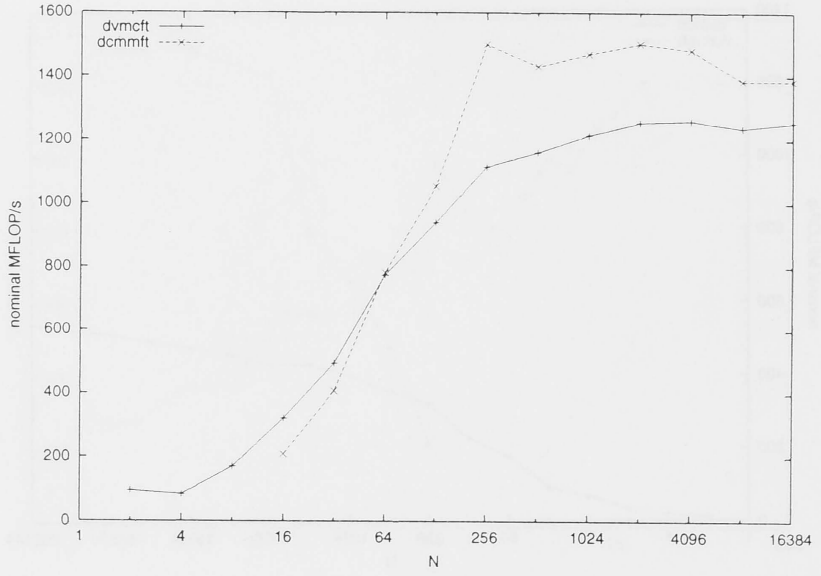


Figure 3.18: Performance comparison of multidimensional FFT algorithms for FFT of size $256 \times N$.

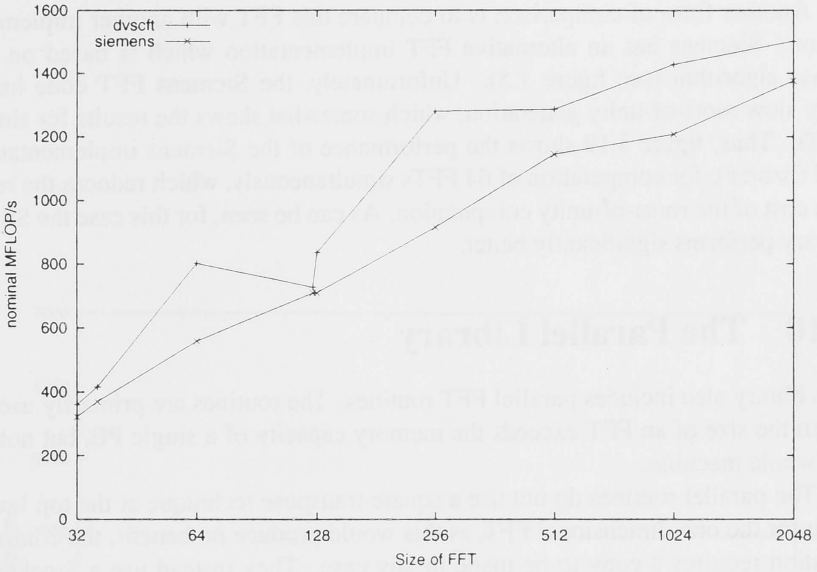


Figure 3.19: Performance comparison of one-dimensional FFT implementations for computation of 64 simultaneous FFTs.

Figures 3.14, 3.15, and 3.16 compare the performance of the two algorithms in the three-dimensional case. Figures 3.17 and 3.18 are similar for the two-dimensional case. The graphs all show 'nominal MFLOPS', which is computed by dividing $5n \log_2 n$ by the time taken for the computation, where n is the total size of the FFT.

As can be seen, the new algorithm, as implemented in `dcmmft`, is consistently faster for both long and intermediate-length FFTs.

Figure 3.16 is of particular interest, as this shows the case of a three-dimensional FFT where all the dimensions are equal. Again, the new algorithm has a significant advantage. Unfortunately, the current implementation of `dcmmft` cannot execute such FFTs when N is not a power of 2 (and, indeed, cannot do this in principle for many sizes).

It is also important to note that the new algorithm requires about half the memory of a non-square transpose routine. This is particularly significant when the FFT being performed is so large that it must be executed out-of-core; in that case, the memory requirements of the in-core FFT is an important determinant of speed.

Another form of comparison is to compare this FFT with another implementation. Siemens has an alternative FFT implementation which is based on the Pease algorithm (see figure 1.5). Unfortunately, the Siemens FFT code has a very slow roots-of-unity generation, which somewhat skews the results for single FFTs. Thus, figure 3.19 shows the performance of the Siemens implementation and `dvscft` for computation of 64 FFTs simultaneously, which reduces the relative cost of the roots-of-unity computation. As can be seen, for this case the SSLII library performs significantly better.

3.10 The Parallel Library

The library also includes parallel FFT routines. The routines are primarily useful when the size of an FFT exceeds the memory capacity of a single PE, but not of the whole machine.

The parallel routines do not use a square transpose technique at the top level, even for the one-dimensional FFT, as this would produce no benefit; the communication requires a copy to be made in any case. They instead use a Stockham FFT, as shown in figure 1.6. The data is arranged so that the low bits of the vector index determine the processor number; then the two FFTs of the Stockham transform are computed using the square transpose technique, and the transposition is the only interprocessor communication.

To avoid an excessive memory requirement, the roots-of-unity computation is performed as the values are needed during the combined transposition and scaling. As many of these values are used only during this transposition, and not in the smaller FFTs, this is not too expensive.

For the multidimensional FFT, a similar computation is performed except that no componentwise multiplication is necessary.

The speedup attained is surprisingly good considering the relatively low ratio of computation to communication. As figure 3.20 shows, the speedup is approximately linear, and is about 4.5 on seven processors. There is a little extra speedup on eight processors, up to 5.3, because in this configuration load balancing is slightly better and the transposition is slightly more efficient.

By way of comparison, the 2048×2048 square transpose routine on one processor runs at 1550 nominal MFLOPS, so some speedup is attained even when moving from this routine to the parallel routine on two processors.

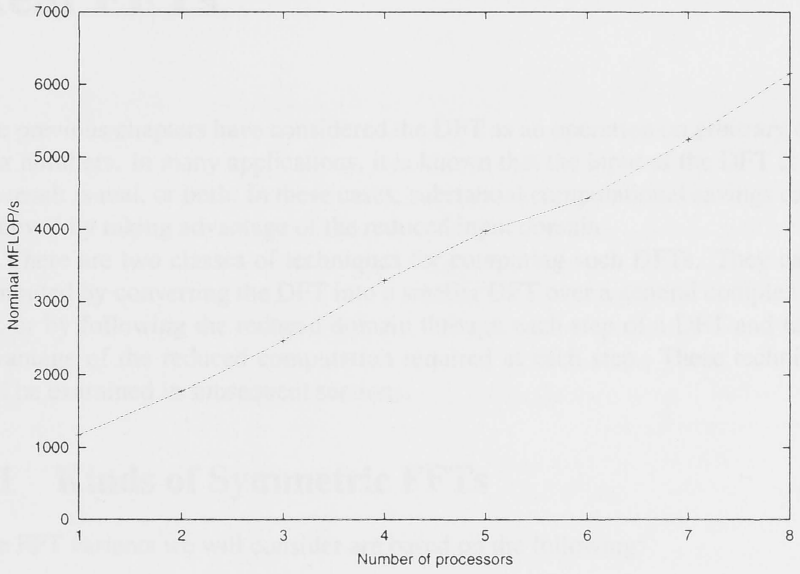


Figure 3.20: Speedup of 2048 × 2048 FFT performed in parallel.

Chapter 4

Real FFTs

The previous chapters have considered the DFT as an operation on arbitrary complex numbers. In many applications, it is known that the input of the DFT is real, the result is real, or both. In these cases, substantial computational savings can be achieved by taking advantage of the reduced input domain.

There are two classes of techniques for computing such DFTs. They can be computed by converting the DFT into a smaller DFT over a general complex vector; or by following the reduced domain through each step of a DFT and taking advantage of the reduced computation required at each step. These techniques will be examined in subsequent sections.

4.1 Kinds of Symmetric FFTs

The FFT variants we will consider are based on the following:

$$x_j = \overline{x_j} \Rightarrow (Fx)_j = \overline{(Fx)_{-j}} \tag{4.1}$$

$$x_j = -\overline{x_j} \Rightarrow (Fx)_j = -\overline{(Fx)_{-j}} \tag{4.2}$$

$$x_j = x_{-j+r} \Rightarrow (Fx)_j = \omega^{qrj}(Fx)_{-j} \tag{4.3}$$

$$x_j = -x_{-j+r} \Rightarrow (Fx)_j = -\omega^{qrj}(Fx)_{-j} \tag{4.4}$$

We will prove the last of these; the remainder are similar and are left to the reader.

$$(Fx)_j = \sum_{k=0}^{n-1} \omega^{qjk}(x)_k \tag{4.5}$$

$$= \sum_{k=0}^{n-1} \omega^{qj(-k+r)}(x)_{-k+r} \tag{4.6}$$

$$= -\sum_{k=0}^{n-1} \omega^{qj(-k+r)}(x)_k \tag{4.7}$$

$$= -\sum_{k=0}^{n-1} \omega^{q(-j)k+qrj}(x)_k \tag{4.8}$$

$$= -\omega^{qrj} \sum_{k=0}^{n-1} \omega^{q(-j)k}(x)_k \tag{4.9}$$

$$= -\omega^{qrj}(Fx)_{-j} \tag{4.10}$$

Note that because of the similarity between forward and inverse DFTs, all of these can be reversed, so that (for instance) if $x_j = \overline{x_{-j}}$, then $(Fx)_j = \overline{(Fx)_j}$. Also note that in the case of symmetries 4.3 and 4.4 when $r = 0$, the forwards and inverse DFTs are identical up to a constant; that is, $F^2x = nx$.

The first two equations give the symmetry in the output of the DFT induced by input data which has zero imaginary part (that is, it is exclusively real) for symmetry 4.1 or zero real part for 4.2. The remaining two apply when there is a reflective symmetry in the data; r determines where the symmetry appears. Depending on whether each of r and n are even or odd, it may happen that up to two values of x are independent of all the other values for symmetry 4.3, or must be set to be zero for 4.4.

4.3 is called an *even* symmetry when $r = 0$ and a *quarter wave even* symmetry when $r = -1$. Similarly, 4.4 is called an *odd* symmetry when $r = 0$, or *quarter wave odd* symmetry when $r = -1$. We will avoid using this terminology, as it leads to confusion; one can have an odd symmetry, an even r and an even n, for example.

Each of the first two symmetries can be combined with either of the last two. The resulting transformations are called the Discrete Cosine Transform, DCT, for combinations of symmetries 4.1 and 4.3 and the Discrete Sine Transform, DST, for combinations of 4.1 and 4.4.

To be precise, what is usually called the DCT is the combination of symmetries 4.1 and 4.3, when n is even and $r = 0$. The usual DST is also when n is even and $r = 0$. There are variants called DCT-II and DST-II for n even and $r = -1$, and some authors add variants DCT-III and DCT-IV (and similarly for DST) for n odd.

The usual DCT and DST can be considered as operations on a real vector x of

length n, so that

$$(DCT(x))_j = \frac{x_0 + x_{n-1}}{2} + \sum_{k=1}^{n-2} x_k \cos \frac{2\pi jk}{2(n-1)} \tag{4.11}$$

$$(DST(x))_j = \sum_{k=0}^{n-1} x_k \sin \frac{2\pi(j+1)(k+1)}{2(n+1)} \tag{4.12}$$

and this is the origin of the names. Note that the corresponding FFT is of length $2(n-1)$ for the DCT and $2(n+1)$ for the DST, and the result of that FFT is twice the result of the DCT and DST.

4.2 Symmetric FFTs Using Complex FFTs

One way to take advantage of the reduced complexity of a symmetric FFT is to apply some pre-processing to combine several symmetric FFTs into a single complex FFT. The general technique to be used is:

1. Given some vectors $x_0, x_1, \ldots, x_{n-1}$ with given symmetries, combine them together to produce some new vector z;
2. Compute Fz;
3. Derive $Fx_0, Fx_1, \ldots, Fx_{n-1}$ from Fz.

For the real FFTs (symmetries 4.1 and 4.2), this is relatively straightforward. Given two vectors x and y, so that x satisfies equation 4.1 and y satisfies 4.2, then we can compute $F(x+y)$ and obtain the components at the end again by noticing that

$$(Fx)_j = \frac{(F(x+y))_j + \overline{(F(x+y))_{-j}}}{2} \tag{4.13}$$

$$(Fy)_j = \frac{(F(x+y))_j - \overline{(F(x+y))_{-j}}}{2} \tag{4.14}$$

If instead we have two real FFTs we can simply compute $F(x+iy)$ instead, and divide iFy by i to obtain the desired result. Similarly, if we have two FFTs with zero real part we can compute $F(-ix+y)$.

Note that the initial calculation here, $x+y$ or $x+iy$ or $-ix+y$, involves no actual computation. This technique seems to have first appeared in [78].

A similar technique can be used to combine the FFTs of two vectors, x with symmetry 4.3 and y with symmetry 4.4, where the same r is used for both: add them together, and then

$$(Fx)_j = \frac{(F(x+y))_j + \omega^{qrj}(F(x+y))_{-j}}{2} \tag{4.15}$$

$$(Fy)_j = \frac{(F(x+y))_j - \omega^{qrj}(F(x+y))_{-j}}{2} \tag{4.16}$$

If we wish to combine two vectors x and z both satisfying 4.3, we can convert z into into a vector y satisfying 4.4 by computing:

$$(y)_k = (z)_{k-1} - (z)_{k+1} \tag{4.17}$$

because then

$$(y)_{-k+r} = (z)_{-k+r-1} - (z)_{-k+r+1} = (z)_{k+1} - (z)_{k-1} = -(y)_k$$

Then note that equation 4.17 is a convolution, so given Fy, we can compute Fz by performing a componentwise division by the DFT of the convolution vector, Fw where $w \star z = y$:

$$(w)_k = \begin{cases} 1, & \text{when } k = 1; \\ -1, & \text{when } k = n-1; \\ 0, & \text{otherwise.} \end{cases} \tag{4.18}$$

That is,

$$(Fz)_k = \frac{\sqrt{-1}(Fy)_k}{2\sin\frac{2\pi qk}{n}} \tag{4.19}$$

It is not necessary to be concerned with potential overflow in the machine arithmetic when $k \neq 0$, as the denominator will be no smaller than $\frac{2}{n}$.

When $k = 0$, $(Fz)_k$ must be computed explicitly as

$$(Fz)_0 = \sum_{j=0}^{n-1} (z)_k \tag{4.20}$$

The same transformation (equation 4.17) can be used to change an z which satisfies symmetry 4.4 into a y which satisfies 4.3.

This allows us to compute four sine or cosine transforms using only a single n-point complex FFT. The techniques presented here have many variations; for

instance, in the previous discussion one can perform the componentwise division first, and apply the convolution to the result of the FFT. Variations of the techniques in the above are presented in [23], although they seem to have been known earlier than this.

Sometimes only a single symmetric FFT need be calculated. In this case, we can use the Stockham algorithm (shown in figure 1.6) to reduce a single FFT into many smaller ones. How precisely this is best done depends on the target of the implementation, but one simple approach, if n is even, is to reduce the single FFT into two or four; then the complex FFT can be computed first, and the remaining order-two or order-four FFT can be combined with the post-processing. A typical example of this may be found in [16] for the order-four case, or in the appendix of [72] for the order-two case.

4.3 Cooley-Tukey for Symmetric FFTs

The technique presented above has several disadvantages. The pre- and post-processing stages involve extra computation, so the speedup obtained is not quite the factor-of-two or factor-of-four one might expect; they also involve additional data accesses, which may be even more expensive than the actual computation. Also, when multiple FFTs are combined, the error in each final result depends on the value of the inputs of all the combined FFTs, which may be undesirable and certainly complicates the error analysis.

A technique which avoids this was first described by Edson and Bergland in 1968, in a pair of patent applications [30] and [11] (the patents expired in 1989), and a paper [10]; the technique is usually attributed to Edson. The technique was implemented in-place in [84] and extended to the other symmetries we consider in [82].

The technique operates by eliminating redundant computations in the computation of a Cooley-Tukey FFT on symmetric data. For instance, consider the case of real data. Consider a Stockham FFT on a vector X of length $st = n$ on such data. Index X by s, t, then:

$$\begin{aligned} Z^{(1)}(\cdot, k) &:= F_s X(\cdot, k) \\ Z^{(2)}(j, k) &:= \omega_n^{jk} Z^{(1)}(j, k) \\ Z^{(3)}(k, j) &:= Z^{(2)}(j, k) \\ Y(\cdot, j) &:= F_t Z^{(3)}(\cdot, j) \end{aligned}$$

The vectors have the following symmetries:

$$\begin{aligned} X(j,k) &= \overline{X(j,k)} \\ Z^{(1)}(j,k) &= \overline{Z^{(1)}(-j,k)} \\ Z^{(2)}(j,k) &= \overline{Z^{(2)}(-j,k)} \\ Z^{(3)}(k,j) &= \overline{Z^{(3)}(k,-j)} \\ Y(k,j) &= \overline{Y(-k,-j)} \end{aligned}$$

The first FFT, F_s, is a real FFT, as its input is real. Thus it can be computed recursively using the technique we are defining. For the remaining computation, it is only necessary to compute one of the values with index j and $-j$ in the above, as the other will be obtained by symmetry; so instead of performing s FFTs of size t, we need only compute a little more than $\frac{s}{2}$ FFTs. Similarly the computation needed in the componentwise multiplication is reduced by almost half.

In the end case, of course, we will eventually have to compute an elementary real FFT of some size to terminate the recursion, just as in the complex case. Such FFTs are usually derived from the corresponding complex FFT routine, either manually or, as in [32], by an automated process.

In total, a little fewer than half the number of arithmetic operations will be required compared to the complex FFT.

The inverse real FFT (that is, one whose result is real) can be obtained by simply reversing the above process.

For the other symmetries, a similar derivation can be made, but the situation becomes more complex.

4.4 Symmetric FFT Library Routines

The SSLII/VP and SSLII/VPP libraries, discussed in chapter 3, also include routines for computing FFTs of real data, and limited routines for computing sine and cosine transforms.

The user interface to the library follows the description in section 3.3. The new routines are called `dvsrft` and `dvmrft` for the real FFTs, `dvmoft` for the cosine transform, and `dvssft` for the sine transform.

The routines all operate by reducing the symmetric FFT to fewer or shorter complex FFTs, then using the existing complex FFT routines. This allows them to take advantage of the significant amount of tuning that has been done on these routines. However, this does add some overhead in the pre- and post- processing

Table 4.1: Pre- and Post- processing routines

Symmetry	pre-processor	post-processor
Real	`dsplit`	`dsepar`
Cosine	`dctc`	`dcte`
Sine	`dstc`	`dste`

which a direct computation would not need. As described below, the cost of this processing on a vector machine is not insignificant.

To obtain better vector length in the one-dimensional case, the real FFT uses an algorithm which is a hybrid of both the algorithms described above.

The names of the routines that perform the pre- and post- processing are listed in table 4.1. The names for the real symmetry are historical; the newer routines have a consistent pattern, of the form 'double-precision cosine transform compression' for `dctc` or '...expansion' for `dcte`.

The sine and cosine transforms are limited in that they support only performing one-dimensional transforms, and require that the number of transforms being performed are a multiple of four.

The real transform, however, is not so limited. Both one- and multidimensional transforms are supported, and speedup is obtained even when performing only a single transform. For the multidimensional transform, the transform is reduced to many one-dimensional real FFTs as in section 2.6, which are then combined pairwise; if the dimensions are odd, one of the FFTs is duplicated to give an even number of real FFTs to be performed, at a slight cost in efficiency.

The one-dimensional FFT is more complex. When a large or an even number of FFTs are to be performed, they are combined pairwise as in the multidimensional real FFT. However, when a small odd number of FFTs are to be performed, a more complex technique is used which is shown in figure 4.1. This technique uses the Stockham algorithm once, as described above, but the size of the FFT is factored into two approximately equal parts. This ensures that the vector length during the data movement shown in the figure is kept at $O(\sqrt{n})$ (although the constant is not as small as might be hoped). The transposition and componentwise multiplication are combined into the routine `dctsv`.

The figure shows the FFT whose input is real; an FFT whose output is to be real (that is, whose input has symmetry $(Fx)_j = \overline{(Fx)_{-j}}$) is obtained by reversing the direction of each arrow, and using inverse routines called `dunfix`, `djoin2`,

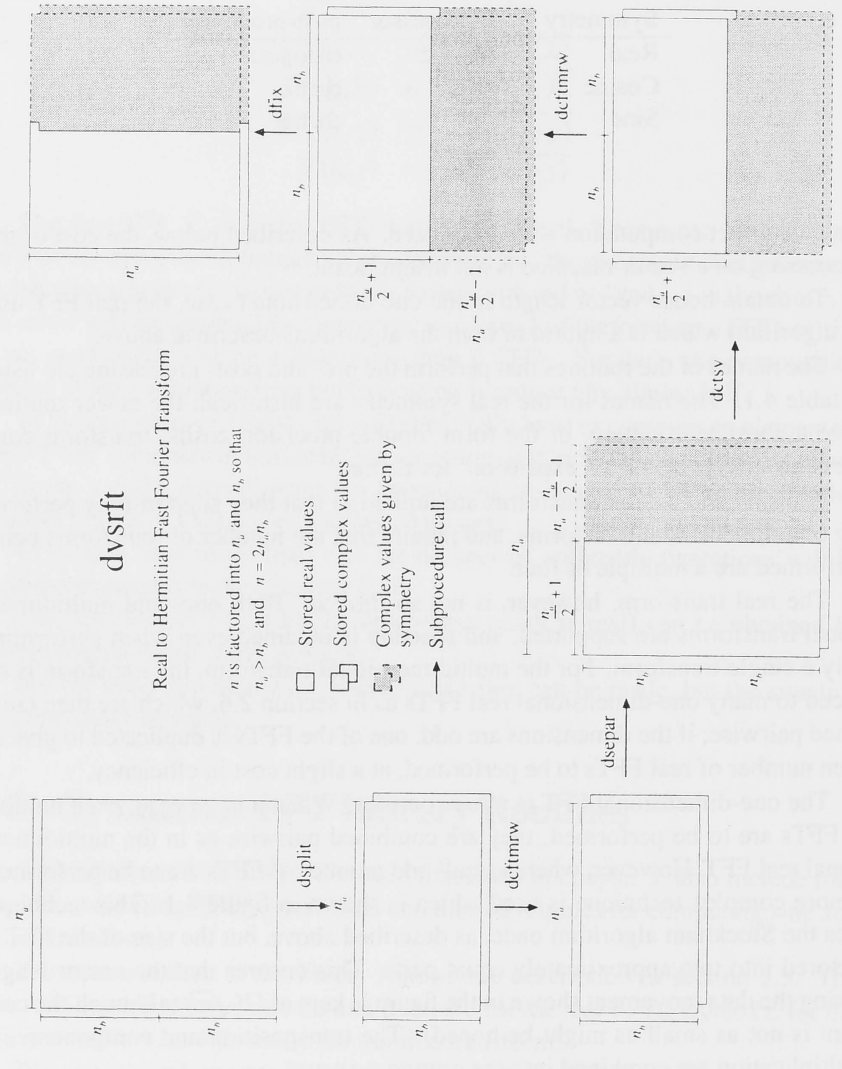


Figure 4.1: Array structure in `dvsrft`.

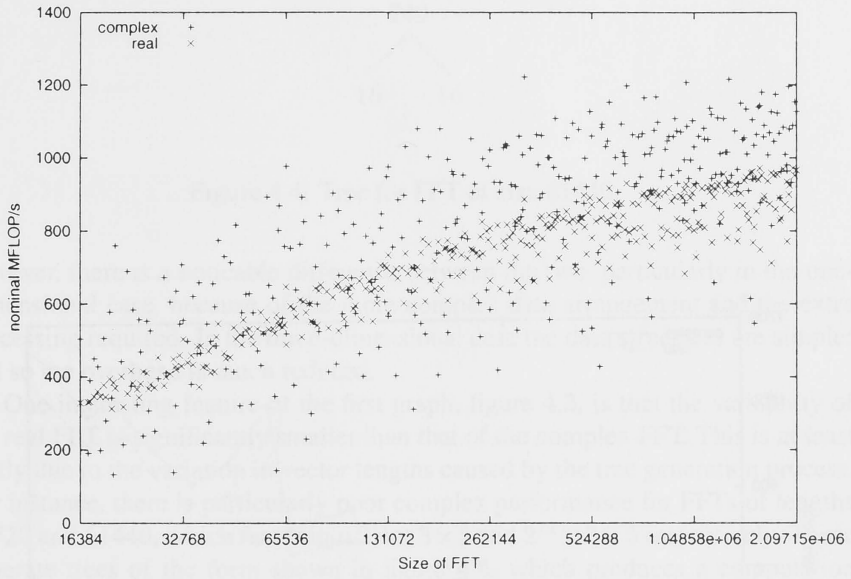


Figure 4.2: Performance of single one-dimensional real FFT.

and `dassem`.

The real FFT uses workspace slightly larger than the input, about $n\sqrt{n}$—the routine's documentation should be consulted to see the exact size—and each step shown takes either the workspace or the input array and copies it into the other array.

4.5 Symmetric FFT Library Performance

Figures 4.2 and 4.3 compare the performance of the real FFT routines (`dvsrft` and `dvmrft`) against that of the complex FFT routines (`dvscft` and `dvmcft`). The 'nominal MFLOP/s' value is computed as $\frac{10^6 t}{2.5 n \log_2 n}$, where t is the time in seconds and n is the number of floating-point values input to the FFT (each complex number is two floating-point values). For the one-dimensional graph, all values between 2^{15} and 2^{21} which factor into powers of 2, 3, and 5 are plotted.

If the real FFTs had no additional overhead over the complex FFTs, we would expect that both would attain about the same 'nominal MFLOP/s'. In practise,

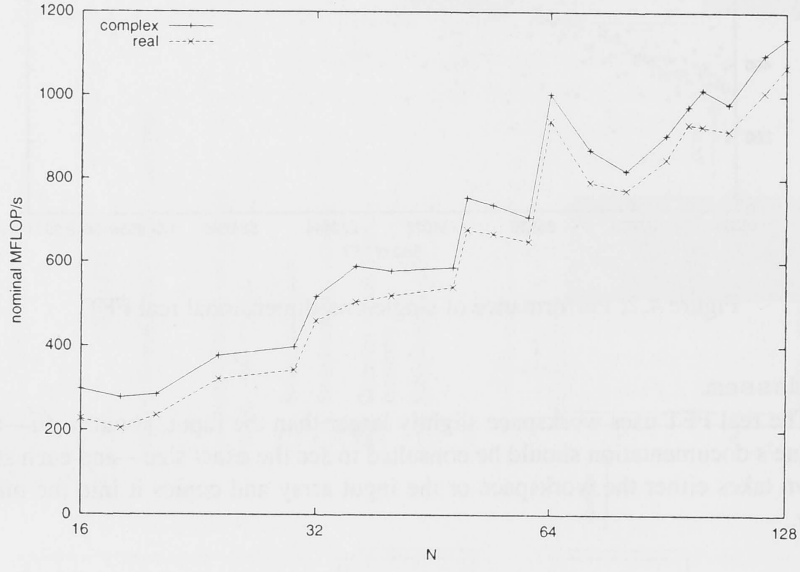


Figure 4.3: Performance of $N \times N \times N$ real FFT.

Figure 4.4: Tree for FFT of size 61440.

however, there is a noticable difference between the two, particularly in the one-dimensional case, because of the more complex data arrangement and the extra processing required. In the three-dimensional case the data structures are simpler and so the overhead is much reduced.

One interesting feature of the first graph, figure 4.2, is that the variability of the real FFT is significantly smaller than that of the complex FFT. This is at least partly due to the variation in vector lengths caused by the tree generation process. For instance, there is particularly poor complex performance for FFTs of lengths 30720 and 61440, which factor into $2^{11} \times 3 \times 5$ and $2^{12} \times 3 \times 5$ respectively. These generate trees of the form shown in figure 4.4, which produces a computation which usually has vector length 16 leading to the poor performance. The real FFT avoids this by using a 'four-step' technique which ensures vector length closer to $\sqrt{n}$, but at the cost of lower *peak* performance, which is probably less useful for users (who can try to arrange their computation to attain the higher performance, or at least avoid the worst cases).

Figure 4.5 shows the performance obtained when executing 1024 sine transforms and compares it to executing 512 real and complex FFTs of the same size; this is how the sine and cosine routines are intended to be used. As can be seen, the performance of the sine transform is similar to that of the real transform, but these are both significantly less than that of the complex transform because of the additional overhead and because the complex transform can usually achieve vector lengths longer than 512 using the 'square transpose' algorithm, but the rate-limiting step in the real and sine transforms, the data rearrangement, does not obtain such benefits.

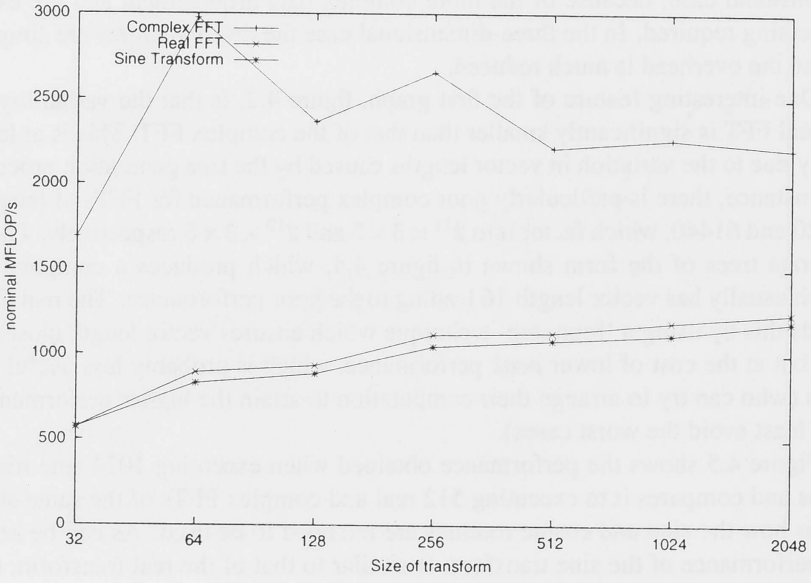


Figure 4.5: Performance of 512-fold one-dimensional transforms.

Bibliography

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-D FFT. *Proc. IEEE*, pages 34–40, 1994.

[2] Ramesh C. Agarwal and James W. Cooley. Vectorized mixed radix discrete fourier transform algorithms. *Proc. IEEE*, 75(9):1283–1292, September 1987.

[3] J. L. Alperin and Rowen B. Bell. *Groups and Representations*, volume 162 of *Graduate Texts in Mathematics*. Springer-Verlag, 1995.

[4] M. Arioli, H. Munthe-Kaas, and L. Waldettaro. Componentwise error analysis for FFT's with applications to fast Helmholtz solvers. Technical Report TR/IT/PA/91/55, Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique, 42 av. G. Coriolis, 31057 Toulouse Cedex, France, 1991.

[5] M. D. Atkinson. The complexity of group algebra computations. *Theoretical Computer Science*, 5:205–209, 1977.

[6] L. Auslander, J. R. Johnson, and R. W. Johnson. Multidimensional Cooley-Tukey algorithms revisited. *Adv. Appl. Math*, 17:477–519, 1996.

[7] Louis Auslander, Ephraim Feig, and Shmuel Winograd. New algorithms for the multidimensional discrete Fourier transform. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-31(2):388–403, April 1983.

[8] Australian National University Supercomputing Facility. *Vectorizing and Tuning for the VPP300*, March 1999.

[9] David H. Bailey and Paul N. Swarztrauber. The fractional Fourier transform and applications. *SIAM Review*, 33(3):389–404, September 1991.

[10] Glenn D. Bergland. A fast Fourier transform algorithm for real-valued series. *Comm. ACM*, 11(10):703–710, October 1968.

[11] Glenn D. Bergland. Fast Fourier transform method and apparatus. US Patent number 3,584,782, July 1968.

[12] D. J. Bernstein. `dbjfft-0.70. ftp://koobera.math.uic.edu/www/djbfft.html`.

[13] Sidney Bertram. On the derivation of the fast Fourier transform. *IEEE Trans. Audio Electroacoustics*, AU-18(1):55–58, March 1970.

[14] J. D. Blanken and P. L. Rustan. Selection criteria for efficient implementation of FFT algorithms. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-30(1):107–109, February 1982.

[15] Leo I. Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Trans. Audio Electroacoustics*, AU-18(4):451–455, December 1970.

[16] Georges Bonnerot. Double odd discrete Fourier transformer. US Patent 4,051,357, March 1976.

[17] William L. Briggs and Van Emden Henson. *The DFT: An Owner's Manual for the Discrete Fourier Transform*. SIAM Publ., 1995.

[18] C. Sidney Burrus. Index mappings for multidimensional formulation of the DFT and convolution. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-25:239–242, 1977.

[19] C. Sidney Burrus. Comments on "selection criteria for efficient implementation of FFT algorithms". *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-31(1):206, February 1983.

[20] C. Sidney Burrus and Peter W. Eschenbacher. An in-place, in-order prime factor FFT algorithm. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-29(4):806–816, August 1981.

[21] R. M. Chamberlain. Gray codes, fast Fourier transforms and hypercubes. *Parallel Computing*, 6:225–233, 1988.

[22] William T. Cochran, James W. Cooley, David L. Favin, Howard D. Helms, Reginald A. Kaenel, William W. Lang, Jr George C. Maling, David E. Nelson, Charles M. Rader, and Peter D. Welch. What is the fast Fourier transform? *IEEE Trans. Audio Electroacoustics*, AU-15(2):45–55, June 1967.

[23] J. W. Cooley, P. A. W. Lewis, and P. D. Welch. The fast Fourier transform algorithm: Programming considerations in the calculation of sine, cosine and Laplace transforms. *J. Sound Vib.*, 12(3):315–337, 1970.

[24] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, 1990.

[26] George Cybenko. Dynamic programming: A discrete calculus of variations. *IEEE Comp. Sci. and Eng.*, pages 92–97, March 1997.

[27] Murray Dow et al. ANU VPP300. `http://anusf.anu.edu.au/VPP/hardware.html`.

[28] M. Drubin. Kronecker product factorization of the FFT matrix. *IEEE Trans. Computers*, C-20:590–593, 1971.

[29] Alan Edelman, Peter McCorQuodale, and Sivan Toledo. The future fast Fourier transform?, March 1997.

[30] James O. Edson. FFT method and apparatus for real valued inputs. US Patent number 3,584,781, July 1968.

[31] D. Fraser. Array permutation by index-digit permutation. *J. ACM*, 23(2):298–309, April 1976.

[32] Matteo Frigo. A fast Fourier transform compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999. to appear.

[33] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, MIT, September 1997.

[34] Carl Friedrich Gauss. Theoria interpolationis methodo nova tractata. In *Werke*, volume 3. Königlichen Gesellschaft der Wissenschaften, Göttingen, 1866.

[35] Izidor Gertner. A new efficient algorithm to compute the two-dimensional discrete Fourier transform. *IEEE Trans. Acoust., Speech, and Signal Process.*, 36(7):1036–1050, July 1988.

[36] S. Godecker. Fast radix 2, 3, 4, and 5 kernels for fast Fourier transformations on computers with overlapping multiply-add instructions. *SIAM J. Sci. Comp.*, 18(6):1605–1611, November 1997.

[37] I. J. Good. The interaction algorithm and practical Fourier analysis. *J.R. Stat. Soc. B*, 20(2):361–372, 1958.

[38] Haitao Guo and C. Sidney Burrus. Fast approximate Fourier transform via wavelets transform. In *Proceedings of SPIE Conference on Mathematical Imaging, Denver CO*, August 1996.

[39] Markus Hegland. A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. *Numerische Mathematik*, 68(4):507–547, 1994.

[40] Markus Hegland. An implementation of multiple and multi-variate Fourier transforms on vector processors. *SIAM J. Sci. Comp.*, 16(2):271–288, March 1995.

[41] Markus Hegland and Wayne Wheeler. Linear bijections and the fast Fourier transform. *Applicable Algebra in Engineering, Communication and Computing*, 8:143–163, 1997.

[42] I. N. Herstein. *Topics in Algebra*. John Wiley & Sons, second edition, 1975.

[43] IEEE. *Proceedings ICASSP 84*, volume 2, San Diego, California, March 1984. IEEE International Conference on Acoustics, Speech, and Signal Processing.

[44] Fujitsu Inc. R&D servers: VX series and VPP300/700 series. `http://www.fujitsu.co.jp/hypertext/Products/Info_process/hpc/vx-e/index-e.html`.

[45] Leah H. Jamieson, Philip T. Mueller Jr, and Howard Jay Siegel. FFT algorithms for SIMD parallel processing systems. *J. Parallel Distributed Comp.*, 3:48–71, 1986.

[46] Judy Jenkinson. Optimising FFT BAFFS on the Fujitsu VPP500. In V. L. Narasimhan, editor, *IEEE First International Conference on Algorithms And Architectures for Parallel Processing*, volume 2, pages 931–932, April 1995.

[47] H. W. Johnson and C. S. Burrus. An in-order, in-place radix-2 FFT. In *Proceedings ICASSP 84* [43], pages 28A.2.1–28A.2.4. IEEE International Conference on Acoustics, Speech, and Signal Processing.

[48] Howard W. Johnson and C. Sidney Burrus. The design of optimal DFT algorithms using dynamic programming. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-31(2):378–387, April 1983.

[49] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. *A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures*. Center for Large Scale Computation, 25 West 43rd St. Suite 400, New York NY 10036, September 1989.

[50] J. R. Johnson and R. W. Johnson. *Challenges of Computing the Fast Fourier Transform*, June 1997.

[51] David K. Kahaner. Matrix description of the fast Fourier transform. *IEEE Trans. Audio Electroacoustics*, AU-18(4):442–450, December 1970.

[52] Margaret Kahn and David Singleton. Personal communication.

[53] Alan H. Karp. Bit reversal on uniprocessors. Submitted for publication, SIAM Review, 1995.

[54] Geoffrey Keating. Choosing trees for FFTs. In *Seventh International Parallel Computing Workshop*, pages P2–X–1–P2–X–6. Australian National University, September 1997.

[55] Geoffrey Keating. Multidimensional vector ffts using only square transpositions. In *Third High Performance Computing Asia Conference and Exhibition*, pages 1285–1289, September 1998.

[56] Geoffrey Keating. The six-step algorithm for two-dimensional FFTs. In John Noye, Michael Teubner, and Andrew Gill, editors, *Computational Techniques and Applications: CTAC97*, pages 345–351, Adelaide, Australia, 1998. The University of Adelaide, World Scientific.

[57] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, third edition, 1997.

[58] Dean P. Kolba and Thomas W. Parks. A prime factor FFT algorithm using high-speed convolution. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-25(4):281–294, August 1977.

[59] Joseph-Louis Lagrange. Recherches sur la nature et la propagation du son. *Miscellanea Taurinensia (Mélanges de Turin)*, I(I-X):1–112, 1759. Reprinted in Oeuvres de Lagrange, Vol. I, J. A. Serret, ed., Paris, 1876, pp. 39–148.

[60] Elliot Linzer and Ephraim Feig. Modified FFTs for fused multiply-add architectures. *Math. Comp.*, 60(201):347–361, January 1993.

[61] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*, volume 10 of *Frontiers in Applied Mathematics*. SIAM Publ., 1992.

[62] David K. Maslen and Daniel N. Rockmore. Generalized FFTs—a survey of some recent results. Technical Report PCS-TR96-281, Dartmouth College, Hanover, New Hampshire USA, 1996.

[63] Russell M. Mersereau and Dan E. Dudgeon. Two-dimensional digital filtering. *Proc. IEEE*, 63(4):610–623, April 1975.

[64] Russell M. Mesereau. A unified treatment of Cooley-Tukey algorithms for the evaluation of the multidimensional DFT. *IEEE Trans. Acoust., Speech, and Signal Process.*, 1981.

[65] Douglas Miles. Compute intensity and the FFT. In *Proceedings of the Conference on Supercomputing '93*, pages 676–684, 1993.

[66] Jacques Morgenstern. Note on a lower bound of the linear complexity of the fast Fourier transform. *J. ACM*, 20(2):305–306, April 1973.

[67] Hans Munthe-Kaas. Superparallel FFTs. *SIAM J. Sci. Comp.*, 14(2):349–367, March 1993.

[68] Kenji Nakayama. A new discrete Fourier transform algorithm using butterfly structure fast convolution. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-33(4):1197–1208, October 1985.

[69] Hiram Paley and Paul M. Weichsel. *A first course in abstract algebra*. Holt, Rinehart and Winston, 1966.

[70] Marshall C. Pease. An adaptation of the fast Fourier transform for parallel processing. *J. ACM*, 15:252–264, 1968.

[71] W. P. Petersen. *Mixed Radix Fast Fourier Transforms for Vector Computers*. AT&T Bell Laboratories, January 1984.

[72] Lawrence R. Rabiner, Ronald W. Schafer, and Charles M. Rader. The chirp z-transform algorithm and its application. *Bell Sys. Tech. J*, 48(5):1249–1292, May-June 1969.

[73] Charles M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE*, 56:1107–1108, June 1968.

[74] Glenn E. Rivard. Direct fast fourier transform of bivariate functions. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-25(3):250–252, June 1977.

[75] Joseph H. Rothweiler. Implementation of the in-order prime factor transform for variable sizes. *IEEE Trans. Acoust., Speech, and Signal Process.*, ASSP-30(1):105–107, February 1982.

[76] John E. Savage. Space-time tradeoffs in memory hierarchies. Technical Report CS 93-08, Brown University, Department of Computer Science, March 1994.

[77] Ivan W. Selesnick and C. Sidney Burrus. Automatic generation of prime length FFT programs. To appear, IEEE Trans. Signal Processing, September 1995.

[78] Richard C. Singleton. On computing the fast Fourier transform. *Comm. ACM*, 10(10):647–654, October 1967.

[79] Richard C. Singleton. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Trans. Audio Electroacoustics*, AU-17(2):93–103, June 1969.

[80] H. Sloate. Matrix representations for sorting and the fast Fourier transform. *IEEE Trans. Circuits Sys.*, CAS-21:109–116, 1974.

[81] T. G. Stockham. High speed convolution and correlation. In *1966 Spring Joint Computer Conference*, volume 28 of *AFIPS Proc.*, pages 229–233, 1966.

[82] Paul N. Swarztrauber. Symmetric FFTs. *Math. Comp.*, 47(175):323–346, July 1986.

[83] Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5(1-2):197–210, 1987.

[84] Clive Temperton. Fast mixed-radix real Fourier transforms. *J. Comp. Phys.*, 52(2):340–350, 1983.

[85] Clive Temperton. A note on prime factor FFT algorithms. *J. Comp. Phys.*, 52:198–204, 1983.

[86] Clive Temperton. Self-sorting mixed-radix fast Fourier transforms. *J. Comp. Phys.*, 52(1):1–23, 1983.

[87] Clive Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *J. Comp. Phys.*, 58:283–299, 1985.

[88] Clive Temperton. A new set of minimum-add small-n rotated DFT modules. *J. Comp. Phys.*, 75:190–198, 1988.

[89] Clive Temperton. Self-sorting in-place fast Fourier transforms. *SIAM J. Sci. Stat. Comput.*, 12(4):806–823, July 1991.

[90] R. Tolimieri. Multiplicative characters and the discrete Fourier transform. *Adv. Appl. Math*, 7:344–380, 1986.

[91] Richard Tolimieri, Myoung An, and Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Signal Processing and Digital Filtering. Springer-Verlag, 1993.

[92] Charles Tong and Paul N. Swarztrauber. Ordered fast Fourier transforms on a massively parallel hypercube multiprocessor. *J. Parallel Distributed Comp.*, 12:50–59, 1991.

[93] Dwen-Ren Tsai and Michael Vulis. Computing discrete fourier transform on a rectangular data array. *IEEE Trans. Acoust., Speech, and Signal Process.*, 38(2):271–276, February 1990.

[94] Teruo Utsumi, Masayuki Ikeda, and Moriyuki Takamura. Architecture of the VPP500 parallel supercomputer. In *Proceedings of Supercomputing USA '94*, November 1994.

[95] Michael Vulis. The weighted redundancy transform. *IEEE Trans. Acoust., Speech, and Signal Process.*, 37(11):1697–1692, November 1989.

[96] M. Wang and E. B. Lee. 2-d FFT algorithm by matrix factorization in a 2-d space. *Multidimens. Systems Signal Process.*, 5(1):61–74, 1994.

[97] S. Winograd. On computing the discrete Fourier transform. *Math. Comp.*, 32(141):175–199, January 1978.

Appendix A

Notation

The following notation is used throughout this document.

$\alpha, \beta, \ldots$	Real and complex numbers.
π	The ratio of a circle's circumference to its diameter.
ω_n	$\exp\left(\frac{2\pi\sqrt{-1}}{n}\right)$, a nth root of unity.
e	The base of natural logarithms, $\sum_{i=0}^{\infty} \frac{1}{i!}$.
h, i, j, k, l	Index variables.
m	Number of dimensions.
n	Length of vectors; size of arrays.
n_i	Size of ith dimension.
w, x, y, z	Vectors, usually of length n.
$A, B, \ldots$	Matrices.
F_n	The Fourier transform matrix or linear transformation of size n.
G, H	Groups.
I_n	The identity matrix of size n.
$\mathbb{C}$	The field of complex numbers.
$\mathbb{R}$	The field of real numbers.
$\mathbb{Z}_n$	The cyclic group of order n.
$\mathcal{M}_n(F)$	The algebra of $n \times n$ matrices over a field F.
$x \star y$	The convolution of vectors x and y.
$x \odot y$	The componentwise product of vectors x and y.
$M \otimes N$	The tensor product of matrices M and N.
$\overline{\phi}$	The complex conjugate of ϕ; if $\phi = \alpha + \sqrt{-1}\beta$, $\overline{\phi} = \alpha - \sqrt{-1}\beta$.
$\overline{A}, \overline{x}$	The (componentwise) conjugate of A or x.

$(x)_j$	The element of x corresponding to j; when j is an integer, the $j+1$th element of x.
x_j	Either $(x)_j$, or one of a number of vectors $x_0, x_1, \ldots$.
$(A)_j$	The $j+1$th row of A.
$(A)_{j,k}$	The $k+1$th element of the $j+1$th row of A.
$\langle x, y \rangle$	The complex inner product of x and y, $\sum_j x_j \overline{y_j}$.
$Z^{(2)}(a, b, c)$	A vector $Z^{(2)}$, written with three indices a, b, c; a is the most significant index.
$\gcd(p, q)$	The greatest common divisor of p and q.
$\ln x$	The natural logarithm of x.
$\log_n x$	The logarithm of x to the base n, $\frac{\ln x}{\ln n}$.
$G \times H$	The direct product of groups or sets G and H.
1_G	The identity element in group G.
$\lvert G \rvert$	The number of elements in the (finite) set G.

Appendix B

Groups

This appendix summarises some basic notation and results on groups, fields, and modules. Some more extensive references for this material are [3, 42, 69].

A *group* is a pair $(G, \cdot)$, where G is a set and $\cdot$ is an operation so that:

1. For all $a, b \in G$, $a \cdot b$ is defined and $a \cdot b \in G$.
2. For all $a, b, c \in G$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$;
3. There exists some *identity element* $1_G \in G$ so that for all $a \in G$, $a \cdot 1_G = 1_G \cdot a = a$;
4. For all $a \in G$, there exists some *inverse* $a^{-1} \in G$ so that $a \cdot a^{-1} = a^{-1} \cdot a = 1_G$.

Often the $\cdot$ is omitted, so ab is written for $a \cdot b$; 1 is written instead of 1_G where the group involved is clear; and the group is referred to as G when it is clear which operation is meant. If the operation is written as $+$, then the inverse of a is written $-a$ and the identity element is written 0.

If n is an integer and a is a member of a group $(G, \cdot)$, a^n is defined to mean $a \cdot a \cdot a \cdot \ldots \cdot a$ where there are n 'a's in the product. If the operation is written as $+$, then na is used to mean the equivalent thing.

The *order* of a finite group G, written $|G|$, is the number of elements in G. The order of an element $g \in G$, written $o(g)$, is the least positive integer n so that $g^n = 1_G$.

$(H, \cdot)$ is a *subgroup* of a group $(G, \cdot)$ iff H is a subset of G and $(H, \cdot)$ is itself a group.

A group G is *abelian* if for all $a, b \in G$, $ab = ba$.

The (external) *direct product* of two groups $(G, \cdot_G)$ and $(H, \cdot_H)$, which we will write $(G, \cdot_G) \times (H, \cdot_H)$, is $(\{(g, h) : g \in G, h \in H\}, \cdot_{G \times H})$ where

$$(a, b) \cdot_{G \times H} (c, d) = (a \cdot_G c, b \cdot_H d).$$

It can be seen that this is itself a group; the identity is $(1_G, 1_H)$ and the inverse of (g, h) is (g^{-1}, h^{-1}). $G \times H$ is abelian if and only if G and H are both abelian.

The direct product can be extended to any finite number of groups.

When the group operation is being written as $+$, we speak of the *direct sum* instead of direct product, and write it $G \oplus H$.

There is also a concept of an *internal direct product*, in which given two subgroups S, T of a group G we can write any element $g \in G$ as a product $g = st$ for some unique $s \in S$ and $t \in T$. It is equivalent to the external direct product.

A *homomorphism* from a group G to a group H is a function $f : G \to H$ so that for all $a, b \in G$, $f(ab) = f(a)f(b)$. From this it follows that $f(1_G) = 1_H$ and $f(g^{-1}) = (f(g))^{-1}$. The image of G in H is a subgroup of H.

Two groups G and H are *isomorphic* if there is a one-to-one and onto homomorphism f between G and H; f is an *isomorphism*. In some sense, two groups that are isomorphic are the same. We write $G = H$ if G and H are isomorphic.

A subgroup H of G is *normal*, written $H \trianglelefteq G$, if for all $h \in H$ and $g \in G$, $ghg^{-1} \in H$.

One very important family of groups are the *cyclic groups*. The cyclic group of order n, which we will write as $\mathbb{Z}_n$, is the group of integers modulo n under addition. The cyclic groups are abelian.

The *fundamental theorem on finite abelian groups* states that any finite abelian group is isomorphic to a direct product of cyclic groups. For a proof, see [42, pp. 109-111].

An (associative) *ring* is a triplet $(R, +, \cdot)$ so that:

1. $(R, +)$ is an abelian group.
2. For all $a, b \in R$, $a \cdot b$ is defined and $a \cdot b \in R$.
3. For all $a, b, c \in R$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
4. For all $a, b, c \in R$, $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$.

A ring need not have an identity under multiplication; an example of a ring without an identity is the even integers. If a ring does have such an element, we will call it a *ring with* 1; often, the term "ring with unit element" or just "ring

with unit" is used for the same property. A *unit* in a ring is an element α so that there is some β so that $\alpha\beta = \beta\alpha = 1$, and *unit element* is another name for the multiplicative identity.

The equivalent of a normal subgroup for a ring is an *ideal*; it is a subgroup S of a ring R so that for any $s \in S$, $r \in R$, both $rs \in S$ and $sr \in S$.

A *homomorphism* from a ring $(R, +, \cdot)$ to a ring $(S, +, \cdot)$ is a function $f : R \to S$ so that

1. For all $a, b \in R$, $f(a + b) = f(a) + f(b)$; and
2. For all $a, b \in R$, $f(a \cdot b) = f(a) \cdot f(b)$.

A ring isomorphism is a one-to-one, onto, homomorphism.

A *division ring* is a ring in which $(R - \{0\}, \cdot)$ is a group.

A *field* is a ring in which $(R - \{0\}, \cdot)$ is an abelian group.

The *characteristic* of a field F is defined as the least positive integer n so that $n\alpha = 0$ for all $\alpha \in F$, if such an integer exists. If no such integer exists, we say that F is of characteristic 0.

The cyclic groups can be viewed as rings under the usual multiplication and addition, and $\mathbb{Z}_n$ is a field if and only if n is prime.

The complex numbers form a field under the usual addition and multiplication. We will write this field as $\mathbb{C}$. The real numbers also form a field which we will write $\mathbb{R}$.

A (left) *module* over a ring R with 1 is a triplet $(M, +, \cdot)$ so that:

1. $(M, +)$ is an abelian group.
2. For all $\alpha \in R$, $m \in M$, $\alpha \cdot m$ is defined and $\alpha \cdot m \in M$.
3. For all $\alpha \in R$, $m, n \in M$, $\alpha \cdot (m + n) = \alpha \cdot m + \alpha \cdot n$.
4. For all $\alpha, \beta \in R$, $m \in M$, $(\alpha + \beta) \cdot m = \alpha \cdot m + \beta \cdot m$.
5. For all $\alpha, \beta \in R$, $m \in M$, $(\alpha\beta) \cdot m = \alpha \cdot (\beta \cdot m)$.
6. For all $m \in M$, $1_R \cdot m = m$.

Often "module over R" is shortened to R-module.

A *vector space* is a module over a field.

A *submodule* of an R-module $(M, +, \cdot)$ is a triplet $(N, +, \cdot)$ where:

1. $(N, +)$ is a subgroup of $(M, +)$; and

2. For all $\alpha \in R, n \in N, \alpha \cdot n \in N$.

A submodule of a vector space is called a *subspace*; it is also a vector space.

A *homomorphism* from an R-module M to an R-module N is a function $f : M \to N$ so that

1. f is a group homomorphism between $(M, +)$ and $(N, +)$; and

2. For all $r \in R, m \in M, f(rm) = rf(m)$.

A module isomorphism is a one-to-one, onto, homomorphism. A homomorphism between vector spaces is called a *linear transformation*.

The (external) *direct sum* of two R-modules $(M, +, \cdot)$ and $(N, +, \cdot)$, which we will write $(M, +, \cdot) \oplus (N, +, \cdot)$, is produced by extending $(M, +) \times (N, +)$ with an operation $\cdot_{M \oplus N}$ defined by

$$r \cdot_{M \oplus N} (a, b) = (r \cdot_M a, r \cdot_N b).$$

$(M, +, \cdot) \oplus (N, +, \cdot)$ is an R-module.

The notation nM, for a module M and integer n, means the direct sum of n copies of M.

An *algebra* over a field $\mathcal{F}$ is a collection $(A, +, \cdot, \star)$ so that

1. $(A, +, \star)$ is a ring;

2. $(A, +, \cdot)$ is an $\mathcal{F}$-vector space;

3. For all $\lambda \in \mathcal{F}, a, b \in A, (\lambda \cdot a) \star b = \lambda \cdot (a \star b) = a \star (\lambda \cdot b)$.

Again, often both $\star$ and $\cdot$ are omitted so $\lambda \cdot (a \star b)$ is written as simply λab.

The canonical example of an $\mathcal{F}$-algebra is the ring of $n \times n$ matrices over $\mathcal{F}$, which we will write $\mathcal{M}_n(\mathcal{F})$.

An algebra homomorphism is a ring homomorphism and $\mathcal{F}$-linear transformation; an algebra isomorphism is a one-to-one and onto homomorphism.

Appendix C

dcft4 Inner Loop

Here, n0 is the vector length, xr and xi are the real and imaginary parts of the data on which the FFT is being performed, n1 is 4, and n2 is $\frac{n}{4}$ where n is the size of the elementary FFT being performed. This code performs one direction of the transform, the code for the other direction is similar but the stores to j1 and j3 are exchanged.

```
        do i2 = 0,n2-1
!OCL novrec(xr,xi)
          do i0 = 0,n0-1
*
*     computing the pointers
*
               j0 = 1+i0+mod(i2*n1,n1*n2)*n0
               j1 = 1+i0+mod(i2*n1+n2,n1*n2)*n0
               j2 = 1+i0+mod(i2*n1+2*n2,n1*n2)*n0
               j3 = 1+i0+mod(i2*n1+3*n2,n1*n2)*n0
*
*     numerical part
*
               t4 = xr(j0)
               t5 = xr(j2)
               t0 = t4 + t5
               t6 = xr(j1)
               t1 = t4 - t5
               t4 = xr(j3)
```

```
            t2 = t6 + t4
            t3 = t6 - t4

            t5 = t0 + t2
            t4 = xi(j1)
            t6 = t0 - t2
            xr(j0) = t5
            t7 = xi(j3)
            t0 = t6 + t7
            xr(j2) = t6
            t2 = t6 - t7
            t4 = t1 + t2
            t6 = xi(j0)
            t5 = t1 - t2
            xr(j1) = t4
            t7 = xi(j2)
            t1 = t6 + t7
            xr(j3) = t5
            t2 = t6 - t7
            t4 = t1 + t0
            t5 = t1 - t0
            xi(j0) = t4
            t6 = t2 + t3
            xi(j2) = t5
            t7 = t2 - t3
            xi(j3) = t6
            xi(j1) = t7
        end do
    end do
```